# Itseohjautuvien järjestelmien arkkitehtuurit

Miia Ketolainen

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Pro gradu -tutkielma
Ohjaajat: Timo Poranen ja Erkki Mäkinen
Toukokuu 2013

Tampereen yliopisto
Informaatiotieteiden yksikkö
Tietojenkäsittelyoppi
Miia Ketolainen: Itseohjautuvien järjestelmien arkkitehtuurit
Pro gradu -tutkielma, 61 sivua
Toukokuu 2013

---

Tässä tutkielmassa käsittelen sellaisten järjestelmien arkkitehtuureja, joilla on kyky mukautua itsessään tai ympäristössään tapahtuviin muutoksiin joko muuttamalla rakennettaan tai käyttäytymistään. Tällainen järjestelmä voi esimerkiksi optimoida suorituskykyään, toipua virheistä tai suojata itseään ulkoisia uhkia vastaan. Esittelen tutkielmassani sellaisia arkkitehtuurityylejä ja suunnittelumalleja, jotka tukevat erilaisia itseohjautuvia ominaisuuksia. Lisäksi käsittelen itseohjautuvien järjestelmien suunnittelun lähtökohtia.

Avainsanat ja -sanonnat: Itseohjautuvat järjestelmät, arkkitehtuurityylit, suunnittelumallit.

## Sisällys

# 1. Johdanto

Itseohjautuvilla järjestelmillä tarkoitetaan järjestelmiä, jotka pystyvät mukautumaan järjestelmässä itsessään ja suoritusympäristössä tapahtuviin muutoksiin ajon aikana. Tällaisia muutoksia voivat olla esimerkiksi käyttäjän antama syöte tai ulkoisista laitteista tai sensoreista tuleva informaatio. Sellaiset muutokset, joihin järjestelmä ei ole varautunut, aiheuttavat lisäkustannuksia virheiden etsimisen, korjaamisen ja järjestelmän uudelleenasentamisen muodossa. Järjestelmän itseohjautuvuus vähentää näitä kustannuksia ja samalla säästää aikaa.

Itseohjautuvuutta suunnitellessa on otettava huomioon järjestelmän tavoitteet ja vaatimukset, järjestelmän itseohjautuvaan käyttäytymiseen johtavat tekijät, mekanismit, joilla itseohjautuvuus saavutetaan sekä itseohjautuvuuden vaikutukset järjestelmään [Cheng et al., 2009].

Itseohjautuvissa järjestelmissä on usein kaksi tai useampia alijärjestelmiä, joista tässä työssä käytetään termejä ohjaava alijärjestelmä ja ohjattava alijärjestelmä. Ohjattavissa järjestelmissä on koko järjestelmän toiminnallisuus, kun taas ohjaavien järjestelmien tehtävänä on tarkkailla ympäristöä sekä ohjattavia järjestelmiä, analysoida mukautumisen tarve, suunnitella mukautumisprosessi ja lopulta antaa ohjattavalle järjestelmälle käsky mukautua. Ohjaavaa alijärjestelmää kutsutaan myös ohjaussilmukaksi. Sen jokaisella komponentilla on tietty tehtävä, ja komponentit kommunikoivat keskenään käyttäen hyväkseen toistensa tuottamaa informaatiota pystyäkseen mukauttamaan ohjattavaa alijärjestelmää.

Tutkielmani tarkoituksena on kartoittaa kirjallisuuden avulla, mitä arkkitehtuurityylejä ja suunnittelumalleja voidaan käyttää itseohjautuvien järjestelmien toteuttamisessa ja mitä itseohjautuvuuteen liittyviä ominaisuuksia niillä voidaan toteuttaa. Lisäksi tarkastellaan ohjausmekanismeja sekä erilaisia tapoja toteuttaa ohjaava alijärjestelmä.

Luvussa 2 käsittelen itseohjautuvia järjestelmiä ja niiden ominaisuuksia ja suunnittelua. Luvussa 3 esittelen UML:n ne kaaviotyypit, joita käytetään itseohjautuvien järjestelmien arkkitehtuurien kuvaamisessa. Luvussa 4 esittelen arkkitehtuurityylejä, joita voidaan käyttää pohjana itseohjautuvien järjestelmien suunnittelussa. Useissa yleisesti käytössä olevissa arkkitehtuurityyleissä on piirteitä, joita voidaan hyödyntää erilaisten itseohjautuvien ominaisuuksien toteuttamisessa. Luvussa 5 esittelen suunnittelumalleja, jotka tukevat järjestelmien itseohjautuvien ominaisuuksien suunnittelua. Suunnittelumallit jaetaan ryhmiin sen mukaan, minkälaisia ominaisuuksia niiden avulla voidaan toteuttaa. Luvussa 6 käsittelen ohjaussilmukoita ja esittelen erilaisia tapoja niiden toteuttamiseksi. Luvussa 7 esittelen erään itseohjautuvan järjestelmän arkkitehtuuria tarkemmin. Luku 8 sisältää yhteenvedon aiheesta sekä ajatuksia siitä, miten aihetta tulisi jatkotutkimuksissa käsitellä.

# 2. Itseohjautuvat järjestelmät

Tässä luvussa käsittelen itseohjautuvia järjestelmiä sekä niiden ominaisuuksia ja suunnittelua. Kohdassa 2.1. esittelen kolme konkreettista esimerkkiä itseohjautuvista järjestelmistä. Kohdassa 2.2. käsittelen itseohjautuvien järjestelmien ominaisuuksia ja määrittelen, mitä milläkin ominaisuudella tarkoitetaan. Kohdassa 2.3. käsittelen itseohjautuvien järjestelmien suunnittelua sekä suunnittelun lähtökohtia ja kohdassa 2.4. esittelen erilaisia lähestymistapoja itseohjautuvien järjestelmien suunnitteluun.

## 2.1. Esimerkkejä itseohjautuvista järjestelmistä

Itseohjautuvaa toiminnallisuutta voi olla monenlaisissa järjestelmissä. Vromant ja muut [2011] käyttävät esimerkkinä liikenteenvalvontajärjestelmää, jossa on itsekorjautuvaa toiminnallisuutta, Cheng ja muut [2009] havainnollistavat työtään miehittämättömän kulkuneuvon avulla, ja Ramirez ja Cheng [2010] ovat soveltaneet itseohjautuvuutta tukevia suunnittelumalleja uutispalvelun suunnittelussa.

Liikenteenvalvontajärjestelmä koostuu kameroista, jotka on asetettu tasaisin välimatkoin valvottavan tien varteen. Jokaisen kameran valvoma alue on rajallinen ja ne on sijoitettu siten, että eri kameroiden valvomat alueet menevät mahdollisimman vähän päällekkäin. Kamerat seuraavat ja tarkkailevat liikenneruuhkia. Kun liikenneruuhkaa ei ole, jokainen kamera muodostaa yksijäsenisen organisaation. Kun liikenne ruuhkautuu siten, että ruuhka ulottuu useamman vierekkäisen kameran valvomille tieosuuksille, näiden kameroiden organisaatiot sulautuvat yhdeksi organisaatioksi. Organisaatiot toimivat Master-Slave -mallin mukaisesti. Yksi organisaation kameroista on Master-komponentti ja muut ovat Slave-komponentteja. Mikäli joku järjestelmän kameroista hajoaa, järjestelmä ei kaadu, vaan kykenee jatkamaan toimintaansa johdonmukaisesti. Mikäli hajonnut kamera oli oman organisaationsa Master-komponentti, jäljelle jääneet kamerat valitsevat keskuudestaan sille korvaajan. Tämän järjestelmän arkkitehtuuri kuvataan tarkemmin luvussa 7. [Vromant et al., 2011]

Miehittämätön kulkuneuvo on auto, jolla ei ole kuskia. Kohdatessaan esteen, kuten ihmisen tai eläimen, se kiertää sen välttääkseen yhteentörmäyksen. Autossa on itsenäinen ohjausjärjestelmä, joka tarkkailee liikennettä ja ympäristöä. Jos se huomaa edessä esteen, se ohjaa auton esteen ohi. [Cheng et al., 2009]

Itseoptimoituva uutispalvelin vastaa asiakkaiden pyyntöihin lähettämällä HTML-dokumentin. Sisältö voidaan lähettää joko tekstimuodossa tai graafisessa muodossa. Vastauksen pitäisi aina olla laadultaan mahdollisimman hyvä, eli mikäli muut vaatimukset sallivat sen, vastaus lähetetään graafisessa muodossa. Vasteaikojen tulee myös pysyä riittävän pieninä, eivätkä kustannukset saa nousta liian suuriksi. Näiden vaatimusten mukaan järjestelmä optimoituu jatkuvasti siten, että vasteajan kasvaessa ja kustannusten salliessa järjestelmä ottaa käyttöön uuden palvelimen. Samoin mikäli vasteaika kasvaa, mutta kustannukset ylittyvät, järjestelmä voi siirtyä lähettämään sisällön teksti-

muodossa graafisen muodon sijaan. Mikäli budjetti on vaarassa ylittyä ja vasteaika on lyhyt, järjestelmä voi poistaa yhden palvelimen käytöstä. [Ramirez and Cheng, 2010]

### 2.2. Itseohjautuvien järjestelmien ominaisuudet

Itseohjautuviksi järjestelmiksi kutsutaan sellaisia järjestelmiä, jotka pystyvät mukautumaan ympäristössään tai järjestelmässä itsessään tapahtuviin muutoksiin ilman, että järjestelmän toiminta häiriintyy. Salehie ja Tahvildari [2009] esittävät itseohjautuvuuden hierarkkisena käsitteenä, joka koostuu kolmesta tasosta: yleisestä tasosta, päätasosta ja primitiivisestä tasosta. Yleisen tason ominaisuuksia ovat itseohjautuvuus ja itseorganisoituvuus. Itseohjautuvuuden käsitteen alle luetaan kuuluvaksi sellaiset itse-etuliitteelliset ominaisuudet kuin *itsejohtavuus* (self-managing), *itsehallitsevuus* (self-governing), *itseylläpito* (self-maintenance), *itsevalvonta* (self-control) ja *itsearviointi* (self-evaluating). Itseorganisoituviksi sanotaan järjestelmiä, joissa hajauttamisella ja uusilla toiminnoilla on suuri merkitys.

Päätasolle kuuluu Salehien ja Tahvildarin [2009] mukaan neljä ominaisuutta, *itsekonfiguroituvuus* (self-configuring), *itsekorjautuvuus* (self-healing), *itseoptimoituvuus* (self-optimizing) ja *itsesuojautuvuus* (self-protecting). Itsekonfiguroituvuudella tarkoitetaan järjestelmän kykyä uudelleenkonfiguroitua automaattisesti asentamalla, päivittämällä ja yhdistämällä ohjelmiston osia. Itsekorjautuvuus määritellään järjestelmän kyvyksi huomata, tunnistaa ja reagoida itsessään tapahtuviin virheisiin ja mukautua siten, että järjestelmä pysyy toimintakykyisenä. Itseoptimoituvuus tarkoittaa järjestelmän kykyä hallita suorituskykyään ja resurssien käyttöä täyttääkseen käyttäjien vaatimukset. Itsesuojautuvuus on järjestelmän kyky havaita tietoturvauhkia ja toipua niiden vaikutuksista.

Primitiivisellä tasolla ovat sellaiset ominaisuudet kuin *itsetietoisuus* (self-awareness), *itsetarkkaileminen* (self-monitoring), *itsesijainti* (self-situated) ja *kontekstitietoisuus* (context-awareness). Itsetietoisuudella tarkoitetaan sitä, että järjestelmä on tietoinen omasta tilastaan ja käyttäytymisestään. Tämä ominaisuus liittyy hyvin vahvasti itsetarkkailemiseen, joka määritellään järjestelmän kyvyksi tarkkailla toimintaansa. Kontekstitietoisuus tarkoittaa järjestelmän kykyä olla tietoinen ympäristönsä tilasta.

Eri tutkijoilla on erilaisia näkemyksiä siitä, mitkä ominaisuudet kuuluvat kunkin käsitteen alle. Esimerkiksi Mühlin ja muiden [2007] mukaan itsejohtavuus on itseorganisoituvuuden yläkäsite. He määrittelevät käsitteiden hierarkian siten, että *itsejohdettavat* (self-manageable) järjestelmät ovat itseohjautuvien järjestelmien osajoukko, jonka osajoukko puolestaan ovat *itsejohtavat* järjestelmät (self-managing). Itsejohtavia ovat sellaiset järjestelmät, joilla on keskitetty ohjauskomponentti, jonka poistamisesta seuraa, että järjestelmä menettää mukautumiskykynsä. *Itseorganisoituvat* järjestelmät (self-organizing) puolestaan ovat itsejohtavien järjestelmien sellainen osajoukko, joilla ei ole keskitettyä ohjauskomponenttia.

Tässä tutkielmassa itseohjautuvuus kuitenkin käsittää kaikki edellä mainitut ominaisuudet. Ollakseen itseohjautuva järjestelmän tulee kyetä tarkkailemaan itseään tai ympäristöään tai molempia.

Tämä ei kuitenkaan vielä riitä, vaan järjestelmän täytyy lisäksi kyetä muuttamaan käyttäytymistään tai rakennettaan oman tai ympäristönsä tilan mukaan. Itseohjautuvalla järjestelmällä on siis jompikumpi tai molemmat ominaisuuksista itsetarkkaileminen ja ympäristön tarkkaileminen. Mikäli järjestelmä on itsetarkkaileva, seuraa siitä, että järjestelmä on itsetietoinen. Vastaavasti kyvystä tarkkailla ympäristöä seuraa kontekstitietoisuus.

Ollakseen itsekorjautuva, järjestelmällä tulee olla neljä ominaisuutta: vian ennustaminen ja havaitseminen, virheen diagnosoiminen tai paikallistaminen, virheen eristäminen tai viasta toipuminen ja validointi. [Gorla, 2007]

Korjausmekanismit käynnistyvät, kun järjestelmässä tapahtuu jokin virhe. Jotta mekanismit alkavat toimia, järjestelmän täytyy joko kyetä ennustamaan vian tapahtuminen tai havaitsemaan tapahtunut virhe, jotta se voi mukautua siten, että se pysyy toimintakykyisenä. Pystyäkseen huomaamaan tapahtumassa olevan tai jo tapahtuneen virheen järjestelmän täytyy pystyä havaitsemaan poikkeavuudet käyttäytymisessään. Sen tulee siis havainnoida itseään ja tunnistaa normaalista poikkeava käytös keräämässään informaatiossa. [Gorla, 2007]

Jotta virheistä toipuminen olisi mahdollista, järjestelmän täytyy pystyä paikallistamaan virheen aiheuttaja. Paikallistamisen ei välttämättä tarvitse onnistua kovin suurella tarkkuudella, vaan järjestelmä voi esimerkiksi tunnistaa komponentin, jossa virhe tapahtui. Järjestelmän tulee siis olla tietoinen omasta tilastaan sekä komponenttien ja niitä pienempien ohjelmistoyksiköiden tiloista. [Gorla, 2007]

Diagnosoituaan vian tai paikallistettuaan virheen aiheuttajan korjausmekanismit voivat joko yrittää poistaa virheen tai eristää sen niin, että se ei pysty aiheuttamaan vahinkoa. Voidakseen eristää vian aiheuttajan ja toipua sen seurauksista järjestelmän pitää pystyä muokkaamaan suoritustaan muuttamalla joko tilojen järjestystä tai lauseiden suoritusjärjestystä tai molempia. Muutokset voivat olla joko pysyviä tai väliaikaisia ja ne voivat tapahtua eri tasoilla, kuten komponenttitasolla tai metoditasolla. [Gorla, 2007]

Kun vian aiheuttaja on poistettu, järjestelmän täytyy varmistaa, että korjaustoimenpiteiden jälkeen järjestelmä täyttää edelleen sille asetetut toiminnalliset ja ei-toiminnalliset vaatimukset. Järjestelmän tulee siis olla myös itsearvioituva. [Gorla, 2007]

Itseorganisoituviksi sanotaan sellaisia järjestelmiä, jotka toimivat ilman keskitettyä ohjauskomponenttia. Sen sijaan niiden toiminta perustuu komponenttien väliseen paikalliseen vuorovaikutukseen. Jokainen komponentti suorittaa omaa tehtäväänsä itsenäisesti ja itseorganisoituvuus syntyy komponenttien välisestä vuorovaikutuksesta. [Serugendo et al., 2004]

### 2.3. Itseohjautuvien järjestelmien suunnittelu

Anderson ja muut [2009] käsittelevät itseohjautuvien järjestelmien mallintamista. He jakavat mallintamisen neljään eri osa-alueeseen: järjestelmän tavoitteet, muutos, mekanismit ja vaikutukset. Tavoitteilla tarkoitetaan koko järjestelmän tavoitteita, muutoksella tarkoitetaan niitä syitä, jotka

aiheuttavat järjestelmän mukautumisen, mekanismeilla tarkoitetaan sitä, kuinka järjestelmä reagoi muutoksiin ja vaikutuksilla sitä, miten mukautuminen vaikuttaa järjestelmään.

Järjestelmän tavoitteita mallinnettaessa on tärkeää ottaa huomioon viisi eri näkökulmaa: evoluutio, joustavuus, kesto, moninaisuus ja riippuvuus. Evoluutionäkökulmalla tarkoitetaan sitä, että on mietittävä, voivatko järjestelmän tavoitteet muuttua sen elinkaaren aikana. Järjestelmän kehittyessä sen tavoitteet saattavat muuttua, niitä voi tulla lisää tai ne voivat poistua. Joustavuusnäkökulma tarkoittaa sitä, miten joustavasti tavoitteet on määritelty, kuinka paljon niihin liittyy epävarmuutta ja ovatko ne tarkasti määriteltyjä, rajoitettuja vai täysin rajoittamattomia. Kestonäkökulmalla tarkoitetaan sitä, onko tavoite validi järjestelmän koko elinkaaren ajan. Tavoite voi olla joko väliaikainen tai pysyvä. Moninaisuusnäkökulma liittyy siihen, kuinka paljon järjestelmällä on itseohjautuvuuteen liittyviä tavoitteita. Niitä voi olla yksi tai useita. Riippuvuusnäkökulmalla tarkoitetaan sitä, että on otettava huomioon mahdolliset suhteet tavoitteiden välillä. Tavoitteet voivat vaikuttaa toisiinsa tai ne voivat olla riippumattomia toisistaan. Toisistaan riippuvat tavoitteet voivat joko täydentää toisiaan tai olla ristiriidassa keskenään. [Anderson et al., 2009]

Järjestelmän muutokseen liittyy neljä eri näkökulmaa: muutoksen lähde, tyyppi, esiintymistiheys ja ennakointi. Muutoksen lähde voi olla joko ulkoinen tai sisäinen. Lähteen ollessa järjestelmän sisäinen on otettava huomioon muutoksen lähteen sijainti: tapahtuuko muutos sovelluksessa itsessään, väliohjelmistossa vai infrastruktuurissa. Muutoksen tyyppi voi olla toiminnallinen, ei-toiminnallinen tai tekninen. Toiminnallinen muutos tarkoittaa sitä, että järjestelmän tarkoitus muuttuu ja sen tuottamien palveluiden täytyy mukautua uuteen tarkoitukseen. Ei-toiminnallinen muutos tarkoittaa, että järjestelmän laatuominaisuus muuttuu. Tekninen muutos tarkoittaa joko ohjelmistossa tai laitteistossa tapahtuvaa muutosta, esimerkiksi väliohjelmiston päivitystä uuteen versioon. Esiintymistaajuusnäkökulmalla tarkoitetaan sitä, kuinka usein tai harvoin muutos tapahtuu. Ennakointinäkökulma tarkoittaa sitä, voidaanko muutos ennustaa etukäteen ja voiko siihen varautua. Muutos voi olla joko odottamaton, ennustettavissa tai odotettavissa. Odottamattomaan muutokseen ei voida varautua, ennustettavissa olevaan muutokseen voidaan varautua ja odotettavissa oleva muutos voidaan ottaa huomioon järjestelmän suunnittelussa. [Anderson et al., 2009]

Itseohjautuvan järjestelmän mekanismien suunnitteluun liittyy seitsemän eri näkökulmaa: mekanismin tyyppi, autonomia, organisointi, laajuus, kesto, ajantasaisuus ja käynnistys. Mekanismin tyypillä tarkoitetaan sitä, liittyykö mukautuminen järjestelmän komponentin parametreihin vai järjestelmän rakenteeseen. Mukautumismekanismi voi siis olla joko rakenteellinen tai parametrillinen. Se voi olla myös näiden kahden tyypin yhdistelmä. Autonomianäkökulmalla tarkoitetaan sitä, liittyykö mukautumismekanismiin ulkopuolisia tekijöitä. Jos mekanismi on täysin autonominen, järjestelmän ulkopuoliset tekijät eivät vaikuta siihen, miten sen täytyy mukautua. Sen sijaan mekanismi on avustettu, jos jokin ulkopuolinen taho, kuten ihminen tai toinen järjestelmä, vaikuttaa siihen. Organisointinäkökulmassa on kyse siitä, onko mukautumismekanismi keskitetty vai hajautettu. Jos mekanismi on keskitetty, siitä on vastuussa yksi komponentti. Jos se on hajautettu, siihen osallistuu useita eri komponentteja. Laajuusnäkökulma liittyy siihen, koskeeko mukautuminen tiettyä osaa

järjestelmästä vai koko järjestelmää. Mukautuminen voi tapahtua joko paikallisesti, jolloin on kyse järjestelmän osasta, tai globaalisti, jolloin on kyse koko järjestelmän laajuisesta mekanismista. Kestonäkökulmassa on kyse siitä, kuinka kauan mukautumisprosessi kestää. Prosessin pituus voi olla lyhyt, keskipitkä tai pitkä, tosin luokittelu on aina riippuvainen järjestelmän sovellusalueesta. Se, mikä yhdessä järjestelmässä on lyhyt aika, voi toisessa olla pitkä. Ajantasaisuusnäkökulmalla tarkoitetaan sitä, miten hyvin voidaan taata mukautumisen tapahtuvan tietyssä ajanjaksossa. Mikäli muutos tapahtuu usein, ei välttämättä voida taata, että edellisen muutoksen laukaisema sopeutumismekanismi on suoritettu loppuun ennen kuin seuraava muutos tapahtuu. Käynnistysnäkökulma tarkoittaa sitä, onko mukautumismekanismin laukaiseva muutos aika- vai tapahtumaperustainen. [Anderson et al., 2009]

Vaikutuksiin liittyviä näkökulmia on neljä: kriittisyys, ennustettavuus, negatiivisuus ja sietokyky. Kriittisyysnäkökulma liittyy siihen, miten mukautumisen epäonnistuminen vaikuttaa järjestelmään. Jotkut vaikutukset voivat olla harmittomia, mutta epäonnistumisesta voi seurata myös kriittisiä ongelmia. Ennustettavuusnäkökulmassa on kyse siitä, missä määrin mukautumisen vaikutuksia voidaan ennustaa. Ennustettavuutta voidaan luonnehtia deterministiseksi tai epädeterministiseksi. Deterministisellä tarkoitetaan sitä, että vaikutukset tiedetään melko tarkasti etukäteen, kun taas epädeterministinen tarkoittaa sitä, että vaikutuksia on hankala ennustaa tarkasti. Negatiivisuusnäkökulmalla tarkoitetaan sitä, millaisia negatiivisia vaikutuksia mukautumisella on järjestelmän suorituskykyyn. Vaikutukset voivat olla merkityksettömiä tai pahimmassa tapauksessa se voi johtaa järjestelmän kaatumiseen. Sietokykynäkökulmassa on kyse siitä, miten hyvin järjestelmä pystyy toimittamaan tarjoamansa palvelut muutosten tapahtuessa. [Anderson et al., 2009]

### 2.4. Näkökulmia itseohjautuvien järjestelmien suunnitteluun

Itseohjautuvien arkkitehtuurien suunnittelua voidaan lähestyä monesta eri näkökulmasta. Esimerkiksi Oreizy ja muut [1999], Weyns ja Holvoet [2007] sekä Hussein ja muut [2012] ehdottavat arkkitehtuuriin perustuvia lähestymistapoja, kun taas Garlan ja Schmerl [2002] esittelevät malliin perustuvan lähestymistavan. Näiden lisäksi järjestelmä voidaan suunnitella myös ei-toiminnallisiin vaatimuksiin perustuen [Kuang and Ormandjieva, 2008; Welsh et al., 2011; Souza, 2012].

Welshin ja muiden [2011] vaatimuksiin perustuva menetelmä on nimeltään REAssuRE (REcording of Assumptions in RE). Siinä mallinnetaan järjestelmän ei-toiminnalliset vaatimukset ns. *pehmeinä vaatimuksina* (softgoal) ja esitetään erilaisten ratkaisustrategioiden vaikutus pehmeiden vaatimusten tyydyttämiseen. *Myötävaikutusyhteyksille* (contribution links), jotka tallettavat tiedon siitä, millainen vaikutus milläkin toteutumisstrategialla on pehmeisiin tavoitteisiin, annetaan arvot, joiden perusteella paras tavoitteidentäyttämisstrategia päätellään. Toteutumisstrategiat mallinnetaan *tehtävinä* (tasks). Tehtävä voi joko *saada aikaan* (make), *auttaa* (help), *vahingoittaa* (hurt), *rikkoa* (break) tai vaikuttaa neutraalisti pehmeän vaatimuksen tyydyttämiseen. Koska aina ei voi varmasti etukäteen tietää, miten erilaiset tavoitteiden toteutumisstrategiat vaikuttavat pehmeisiin vaatimuksiin, suunnittelija voi joutua tekemään olettamuksia. Pehmeiden vaatimusten myötävaikutusyhteyk-

siin liitetään *väitteitä* (claims), jotka tallettavat tiedon siitä, miten tavoitteen toteutumisstrategian valintaan on päädytty, mikäli asiaan liittyy epävarmuutta. Väitteillä merkitään epävarmuutta, mikä auttaa suunnittelijaa arvioimaan, millaisia seurauksia virheellisistä olettamuksista voi aiheutua. Väitteet tekevät siis suunnitteluratkaisujen takana olevat olettamukset implisiittisiksi ja jäljitettäviksi. Niitä voidaan kuitenkin käyttää myös dynaamisesti. Ylläpitämällä tavoitemallien ajonaikaista esitystä ja tarkkailemalla väitteitä kontekstimuuttujien avulla voidaan johtaa virheelliseksi osoittautuvan väitteen vaikutus tavoitemalleihin ja rinnakkaisten tavoitteidentoteutumisstrategioiden edut voidaan arvioida uudelleen dynaamisesti. Siten järjestelmä voi lieventää vastaavaa virheellistä oletusta mukauttamalla itsensä sellaiseen konfiguraatioon, joka vastaa parempaa strategiaa.

Väitteiden johtamisen semantiikka on yksinkertainen. Väitteet joko saavat aikaan tai rikkovat myötävaikutusyhteyden, joihin ne on liitetty:

$$\text{claim} = \text{break} \rightarrow c' = \text{neutral}$$
$$\text{claim} = \text{make} \wedge (c = \text{hurt} \vee c = \text{break}) \rightarrow c' = \text{break}$$
$$\text{claim} = \text{make} \wedge (c = \text{help} \vee c = \text{make}) \rightarrow c' = \text{make}.$$

Arvo c viittaa myötävaikutusyhteyteen, johon väitteet on liitetty, ja c´ on myötävaikutusyhteyden arvo, kun c on yhdistetty väitteen vaikutukseen. Väitteen totuusarvo voi siis vaikuttaa siihen, miten hyvin tehtävä tyydyttää pehmeän tavoitteen.

Souzan [2012] lähestymistapa perustuu tavoitteisiin, pehmeisiin vaatimuksiin, laaturajoitteisiin ja olettamuksiin. Lisäksi ohjaussilmukoilla on tärkeä rooli. Silmukan tarkkailijakomponentti valvoo vaatimusten toteutumista. Samoin kuin Welshin ja muiden [2011] lähestymistapa, myös Souzan [2012] tapa sallii sen, että kaikkia tavoitteita ei välttämättä pystytä täyttämään kaikissa tilanteissa.

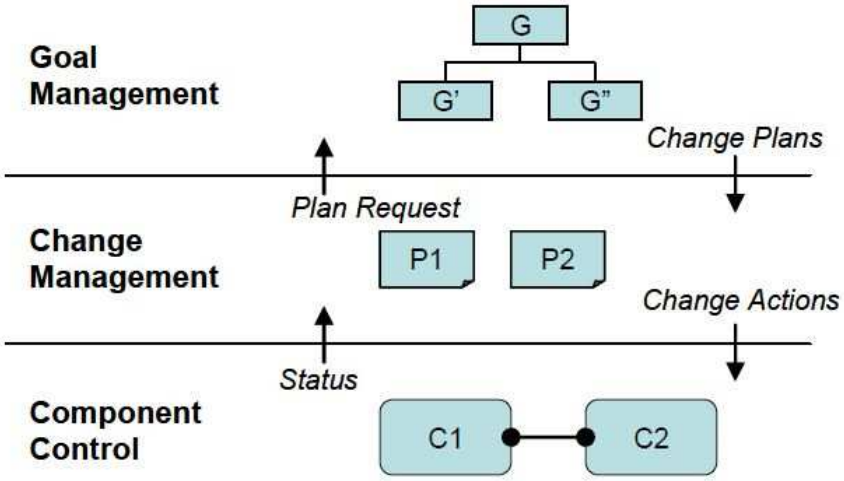
Ensin määritellään *tietoisuusvaatimukset* (awareness requirements, AwReqs). Tietoisuusvaatimuksilla tarkoitetaan sellaisia vaatimuksia, jotka määräävät muiden vaatimusten ajonaikaisen tilan, esimerkiksi onnistumisen tai epäonnistumisen. Tietoisuusvaatimukset muodostavat vaatimukset ohjaussilmukoille, jotka toteuttavat järjestelmän itseohjautuvat ominaisuudet. Ne siis kuvaavat ei-toivotut tilanteet, joihin järjestelmän pitää sopeutua mikäli ne toteutuvat. Tietoisuusvaatimus voi esimerkiksi määrätä, kuinka usein jokin tavoite saa epäonnistua tai kuinka usein sen täytyy onnistua. Ajonaikana tavoitemallin elementit esitetään luokkina, joiden instansseja luodaan joka kerta, kun käyttäjä aloittaa toimenpiteen tavoitteen saavuttamiseksi tai kun tavoitteen onnistuminen pitää muute varmentaa. Tietoisuusvaatimuksia käytetään vaatimusten toteutumisen indikaattoreina ajon aikana. Jos ne epäonnistuvat, järjestelmä etsii sellaisen mukautumisstrategian, joka parantaa vaaditut ominaisuudet. [Souza, 2012]

Joissakin tapauksissa mukautumisratkaisu voi löytyä vaatimusten itsensä joukosta. Tähän tarkoitukseen Souza [2012] esittelee *evoluutiovaatimukset* (evolution requirements, EvoReqs). Evoluutiovaatimuksilla tarkoitetaan määriteltyjä muutoksia, jotka tehdään vaatimusmalliin olio- tai luokkatasolla määrätyissä olosuhteissa. Evoluutiovaatimukset määritellään käyttämällä joukkoa

primitiivisiä operaatioita, jotka suoritetaan mallin elementeille. Jokainen operaatio liittyy sovelluskohtaisiin toimintoihin, jotka järjestelmään toteutetaan. Lisäksi niitä voidaan yhdistää malleja käyttäen mukautumisstrategioiden muodostamiseksi. Nämä mallit liitetään määrättyihin tietoisuusvaatimuksiin tavoitemallissa ja niitä voidaan käyttää ajon aikana ohjaamaan järjestelmän mukautumista.

Kramer ja Magee [2007] esittelevät arkkitehtuuriperustaisen lähestymistavan. Sen mukaan arkkitehtuuriin perustuvalla lähestymistavalla on paljon etuja. Sitä voidaan hyödyntää monien erilaisten järjestelmien suunnittelussa, sillä sen periaatteita ja käsitteitä voidaan soveltaa monenlaisiin sovellusalueisiin. Se on myös riittävän abstrakti kuvaamaan dynaamisia muutoksia järjestelmissä. Eduiksi voidaan lukea myös skaalautuvuus, lukuisat aiemmat aiheesta tehdyt tutkimukset, joihin se perustuu, sekä mahdollisuus yhdistää se muihin lähestymistapoihin.

Kramerin ja Mageen [2007] näkemyksen mukaan itseohjautuvissa järjestelmissä voidaan käyttää abstraktia kolmitasoarkkitehtuuria kuvaamaan järjestelmän rakenne. Arkkitehtuurin kolme kerrosta ovat komponenttien hallinta, muutosten hallinta ja tavoitteiden hallinta. Kuvassa 1 on kuvattu kerrokset ja niiden välinen vuorovaikutus.




Kuva 1. Kolmikerrosarkkitehtuuri. [Kramer and Magee, 2007]

Komponenttien hallinta on kolmikerrosarkkitehtuurin alin kerros. Se koostuu toisiinsa yhteydessä olevista komponenteista, jotka toteuttavat järjestelmän toiminnallisuuden. Lisäksi se sisältää toiminnot, joiden avulla järjestelmän voi raportoida komponenttiensa tilan ylemmille kerroksille ja luoda, poistaa ja yhdistää komponentteja. Alimman kerroksen tehtävänä on siis havaita sellaiset tilanteet, joissa järjestelmän täytyy mukautua, ja raportoida niistä ylemmille kerroksille. [Kramer and Magee, 2007]

Keskimmäinen kerros on muutosten hallinta. Sen tehtävänä on välittää muutokset komponenttienhallintakerrokselle, mikäli kyseinen kerros raportoi uusista tiloista, tai mikäli ylimmältä kerrokselta tulee tieto uusista tavoitteista. Muutostenhallintakerroksessa voidaan myös luoda uusia komponentteja, muuttaa komponenttien välisiä yhteyksiä ja muuttaa komponenttien parametreja. Se

koostuu joukosta ennalta määriteltyjä suunnitelmia, jotka aktivoidaan, mikäli järjestelmän operationaalisessa tilassa tapahtuu muutoksia. Jos järjestelmä päätyy tilaan, jolle ei ole ennalta määritelty mukautumissuunnitelmaa, muutostenhallintakerros ilmoittaa siitä ylimmälle kerrokselle. Mikäli muutostenhallintakerros saa ylemmältä kerrokselta ilmoituksen uusista tavoitteista, niihin liittyy aina myös jokin uusi suunnitelma, joka toteutetaan, jos tavoitteet eivät täyty. [Kramer and Magee, 2007]

Ylin kerros on tavoitteidenhallintakerros. Sen tehtävänä on tuottaa muutostenhallintasuunnitelmia, jos alemmalta kerrokselta tulee pyyntö tai jos järjestelmään lisätään uusia tavoitteita. Suunnitelma voi esimerkiksi kuvata, mitä uusia komponentteja pitää luoda tai mitä komponenttien välisiä yhteyksiä pitää muuttaa, jotta järjestelmän tavoitteet täyttyisivät. [Kramer and Magee, 2007]

Weyns ja Holvoet [2007] esittelevät arkkitehtuuristrategian, jota voidaan käyttää itseohjautuvien järjestelmien suunnittelussa. Se koostuu joukosta malleja, joilla voidaan toteuttaa järjestelmän eri osia ja jotka voidaan integroida yhteen kokonaiseksi järjestelmäksi. Strategia yleistää yhteiset toiminnot ja rakenteet, jotka itseohjautuvissa järjestelmissä esiintyvät sekä tarjoaa uudelleenkäytettävän suunnittelukehyksen järjestelmille, joilla on tietyt yhteiset piirteet. Kehys auttaa kehittämään tällaisia järjestelmiä luotettavammin ja kustannustehokkaammin.

Arkkitehtuuristrategia soveltuu sellaisiin järjestelmiin, jotka ovat dynaamisia, joiden toimintaolosuhteet, kuten palveluiden ja resurssien saatavuus muuttuvat, ja joita on vaikea hallita globaalisti. Tällaisia järjestelmiä ovat esimerkiksi liikenteenvalvontajärjestelmät ja mobiili- ja sensoriverkostot. Ne koostuvat agenteista, jotka on sijoitettu järjestelmän suoritusympäristöön ja jotka keräävät tietoa, jonka ne jakavat toisten agenttien kanssa. [Weyns and Holvoet, 2007]

Arkkitehtuuristrategiassa on kaksi ylimmän tason moduulia, agentti ja sovellusympäristö. Agentti on itsenäinen komponentti, jonka tehtävänä on suorittaa sille asetetut tavoitteet. Se kapseloi oman tilansa ja ohjaa käyttäytymistään itse. Agentit on sijoitettu ympäristöön, jota ne havainnoivat ja jossa ne toimivat vuorovaikutuksessa toistensa kanssa. Sovellusympäristö mahdollistaa agenteille tiedon jakamisen ja käyttäytymisen muuttamisen. Sen päätehtävät ovat tarjota pääsy ulkoisiin resursseihin, mahdollistaa agenttien tarkkailla ja käsitellä ympäristöään ja kommunikoida toisten agenttien kanssa ja toimia välittäjänä agenttien välillä. [Weyns and Holvoet, 2007]

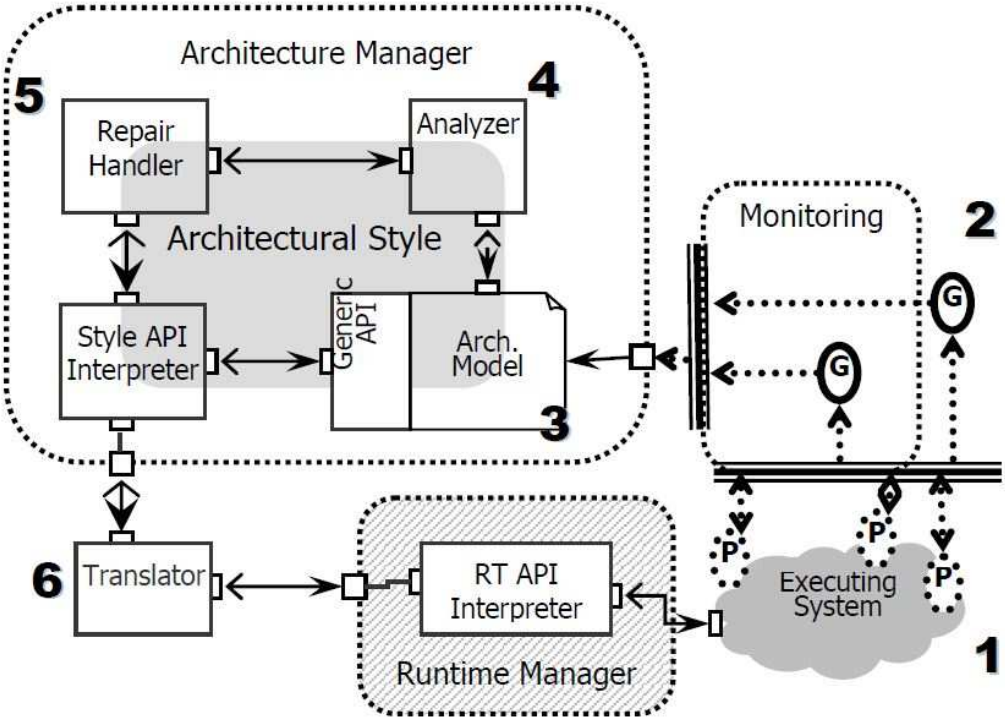
Agentti koostuu kolmesta alikomponentista, tarkkailijakomponentista, päätöksentekijäkomponentista ja viestintäkomponentista. Ne jakavat tiedon järjestelmän kulloisestakin tilasta. Tarkkailijakomponentti on vastuussa tiedon keräämisestä. Se voi suodattaa keräämäänsä tietoa siten, että se lähettää eteenpäin vain sellaista tietoa, joka on tarpeellista sen senhetkiselle tehtävälle. Päätöksentekijäkomponentin tehtävänä on valita, mikä toiminto missäkin tilanteessa suoritetaan. Arkkitehtuuristrategian toimintomalli perustuu vaikutus-reaktio -malliin, joka erottaa toisistaan agenttien tuottamat vaikutukset, ts. yritykset muuttaa tapahtumien kulkua, ja reaktiot, jotka johtavat tilojen muutoksiin. Päätöksentekijäkomponentti valitsee vaikutukset, joilla agenttien tehtävät toteutetaan. Viestintäkomponentin tehtävänä on olla vuorovaikutuksessa muiden agenttien kanssa. Se vaihtaa vieste-

jä muiden agenttien vastaavien komponenttien kanssa jakaakseen tietoa. [Weyns and Holvoet, 2007]

Sovellusympäristö koostuu seitsemästä osakomponentista ja tilatietovarastosta, jonka ne jakavat keskenään. Osakomponentit ovat esitystapageneraattori, tarkkailija- ja tiedonkäsittelijäkomponentti, vuorovaikutuskomponentti, matalan tason ohjauskomponentti, viestinvälittäjä, viestipalvelu ja synkronisointi- ja tiedonkäsittelijäkomponentti. Esitystapageneraattori tarjoaa agenteille toiminnallisuuden, jonka avulla ne voivat tarkkailla ympäristöä. Kun agentti kerää tietoa ympäristöstä, generaattori käyttää sovellusympäristön senhetkistä tilaa tuottaakseen agentille esitystavan. Tarkkailija- ja tiedonkäsittelijäkomponentin tehtävänä on tarjota toiminnallisuus tarkkailla järjestelmän fyysistä ympäristöä, kuten laitteistoa ja kerätä tietoa järjestelmän muilta solmuilta. Vuorovaikutuskomponentti on vastuussa agenttien vaikutuksesta ympäristöön. Agentit voivat vaikuttaa ympäristöönsä kahdella tavalla. Ne voivat joko yrittää muokata sovellusympäristön tilaa tai fyysisen ympäristön resurssien tilaa. Matalan tason ohjauskomponentti muuttaa agenttien aiheuttamat vaikutukset matalan tason primitiivisiksi toiminnoiksi fyysisessä ympäristössä. Viestinvälittäjäkomponentin tehtävänä on välittää viestejä agentilta toiselle. Se kerää viestit, puskuroi ne ja toimittaa ne oikeille vastaanottajille. Viestipalvelun tehtävänä on muuntaa viestit fyysisen ympäristön ymmärtämiksi primitiiveiksi. Synkronisointi- ja tiedonkäsittelijäkomponentti pitää huolta siitä, että sovellusympäristön ja fyysisen ympäristön tilat on synkronoitu keskenään ja että hajautetun järjestelmän kaikkien solmujen sovellusympäristöt on synkronoitu keskenään. [Weyns and Holvoet, 2007]

Garlan ja Schmerl [2002] esittelevät malleihin perustuvan lähestymistavan itseohjautuvuuden suunnitteluun. Sen mukaan järjestelmä ylläpitää yhtä tai useampaa mallia, jotka kuvaavat järjestelmää ajonaikana. Mallilla tarkoitetaan abstraktia kuvausta järjestelmästä. Malli voi kuvata esimerkiksi järjestelmän arkkitehtuuria, suorituskykyä tai luotettavuutta. Lisäksi järjestelmässä on ulkoisia komponentteja, joiden tehtävänä on tarkkailla järjestelmän käyttäytymistä, määrittää, milloin järjestelmän täytyy mukauttaa itseään ja tarvittaessa käynnistää mukautuminen. Malleihin perustuvaa lähestymistapaa perustellaan sillä, että ulkoiset komponentit ovat uudelleenkäytettäviä, koska ne eivät ole sovelluskohtaisia ja ne voidaan helposti vaihtaa, ja tarvittavat mallit voidaan valita sen mukaan, mikä missäkin järjestelmässä on tärkeää.

Malliperustaisessa lähestymistavassa ajonaikaista järjestelmää tarkkaillaan, jotta havaitaan poikkeamat sen käyttäytymisessä. Tarkkailtavat arvot liittyvät järjestelmän arkkitehtuurimallin ominaisuuksiin. Arkkitehtuurimallin ominaisuuksien muuttumisesta seuraa rajoitteiden arvioiminen, jotta voidaan määrittää, toteutuvatko järjestelmälle asetetut laatuvaatimukset riittävän hyvin. Jos raja-arvot ylittyvät, käynnistetään korjausmekanismi, joka mukauttaa arkkitehtuuria. Kuvassa 2 esitetään tällaisen järjestelmän toiminta abstraktilla tasolla.



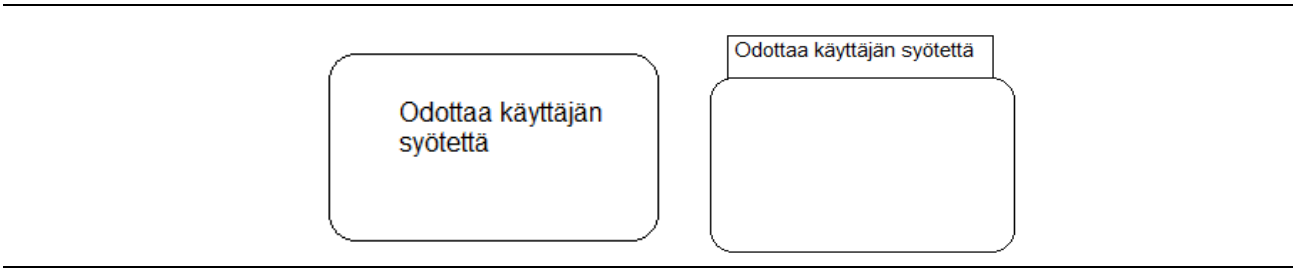

Kuva 2. Malleihin perustuvan itseohjautumisen toimintaperiaate. [Garlan and Schmerl, 2002]

# 3. UML

Unified Modeling Language (UML) on Object Management Groupin (OMG) kehittämä ja ylläpitämä standardi, jolla kuvataan järjestelmien arkkitehtuuria, käyttäytymistä ja rakennetta [OMG, 2006]. UML määrittelee erilaisia kaaviotyyppejä, joilla järjestelmiä kuvataan eri näkökulmista. Nämä kaaviotyypit ovat käyttötapauskaavio, luokkakaavio, sekvenssikaavio, sijoittelukaavio, luokkakaavio, oliokaavio, komponenttikaavio, pakkauskaavio, koostekaavio, aktiviteettikaavio, kommunikointikaavio, kokoava vuorovaikutuskaavio ja ajoituskaavio. Näistä kaaviotyypeistä olennaisia tämän tutkielman kannalta ovat tilakaavio ja luokkakaavio. Tilakaavioita tarkastelen kohdassa 3.1. ja luokkakaavioita kohdassa 3.2.

## 3.1. Tilakaaviot

Tilakaavioilla kuvataan järjestelmän käyttäytymistä. Niissä esitetään järjestelmän tila jonkin prosessin aikana sekä kyseisen prosessin aikaiset siirtymät tilasta toiseen. Prosessi kuvataan pyöristettynä suorakulmiona, jonka sisälle piirretään prosessin aikaiset tilat ja tilasiirtymät. Tila kuvataan suorakulmiona, jonka kulmat ovat pyöristetyt ja jonka sisällä on tilan nimi (kuva 3). Vaihtoehtoisesti tilan voi myös esittää siten, että suorakulmion yläreunaan liitetään laatikko, jossa on tilan nimi. [OMG, 2006]



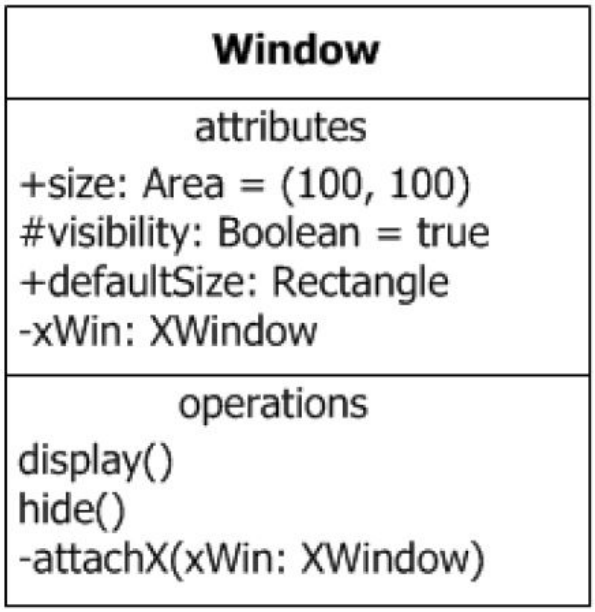

Kuva 3. Tilasymbolin vaihtoehtoiset esitystavat.

Tilasiirtymiä kuvataan nuolilla, jotka lähtevät yhtä tilaa kuvaavasta pyöristetystä suorakulmiosta ja päättyvät seuraavaa tilaa kuvaavaan symboliin. Nuolien ja pyöristettyjen suorakulmioiden lisäksi tilakaaviossa käytetään erilaisia ympyröitä merkitsemään pseudotiloja. Pseudotiloja käytetään yhdistämään ja ohjaamaan tilasiirtymiä. Pseudotiloja ovat tilakaavion alkutila, jota merkitään mustalla ympyrällä, lähihistoria-tila, jota merkitään ympyrällä, jonka sisällä on H-kirjain, syvä historia -tila, jossa H-kirjaimen lisäksi on *-merkki, tulopiste, jota merkitään valkoisella ympyrällä, lähtöpiste, jota merkitään valkoisella ympyrällä, jossa on rasti, valintatila, jota merkitään vinoneliöllä, keskeytystila, jota merkitään rastilla, sekä haara ja liitos, joita merkitään mustalla palkilla. Liitokseen tulee kaksi tilasiirtymää ja siitä lähtee yksi, kun taas haaratilaan tulee yksi siirtymä ja siitä lähtee kaksi siirtymää. Tilakaavion lopputilaa kuvataan ympyrällä, jonka sisällä on musta ympyrä. Sitä ei kuitenkaan UML:n määrittelyssä lasketa pseudotilaksi. [OMG, 2006]

Tilakaavioihin merkitään myös ehdot tilasiirtymien toteutumiselle. Ehto kirjoitetaan lähtötilasta tulotilaan johtavan nuolen viereen. Ehto voi olla esimerkiksi jokin rajoite, kuten kulunut aika, tai jonkin toisen komponentin tietty tila.

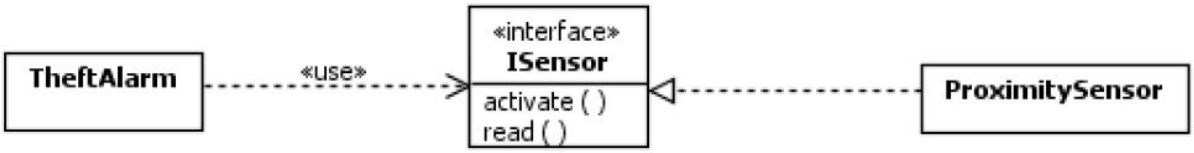
### 3.2. Luokkakaaviot

Luokkakaavion avulla kuvataan järjestelmän rakenne. Siinä esitetään järjestelmän luokat, niiden ominaisuudet ja operaatiot sekä luokkien väliset suhteet. Luokka kuvataan suorakulmiona, joka jaetaan kolmeen osaan (kuva 4). Ylimmässä osassa on luokan nimi, keskimmäisessä luokan ominaisuudet ja niiden tyypit ja alimmassa osassa operaatiot, niiden paluuarvot ja parametrit tyyppeineen. Ominaisuudet esitetään siten, että ensin kirjoitetaan ominaisuuden nimi, sitten kaksoispiste ja lopuksi ominaisuuden tyyppi. Mikäli ominaisuudella on jokin oletusarvo, merkitään tyypin jälkeen yhtäsuuruusmerkki ja ominaisuuden oletusarvo. Operaatiot esitetään siten, että ensin merkitään operaation nimi ja sen jälkeen suluissa parametrien nimet tyyppeineen. Ominaisuuksien ja operaatioiden näkyvyysmääreet merkitään niiden nimien eteen. Public-määre merkitään +-merkillä, private-määre merkitään - -merkillä ja protected-määre merkitään #-merkillä.

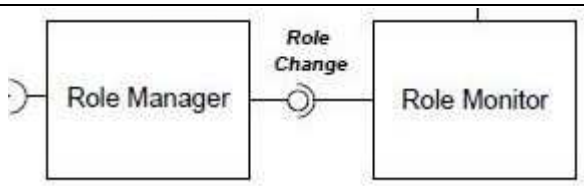



Kuva 4. Luokka kuvattuna UML:lla. [OMG, 2006]

Luokkakaavioihin piirretään myös luokkien toteuttamat rajapinnat. Rajapinnat kuvataan samalla tavalla kuin luokat, suorakulmioina, mutta ylimpään osaan ennen rajapinnan nimeä kirjoitetaan <<interface>> (kuva 5). Rajapinnan operaatiot merkitään samalla tavalla kuin luokkienkin. Rajapinnan toteuttaminen kuvataan siten, että toteuttavasta luokasta piirretään nuoli toteutettavaan rajapintaan. Nuolen kärki on kolmio ja viiva on katkoviiva. Vaihtoehtoisesti rajapinta voidaan merkitä ympyränä, johon tulee viiva sen toteuttavasta luokasta. Luokasta, joka käyttää rajapinnan tarjoamia palveluja, piirretään kaareen päättyvä viiva kohti rajapintaa kuvaavaa ympyrää kuvan 6 osoittamalla tavalla.

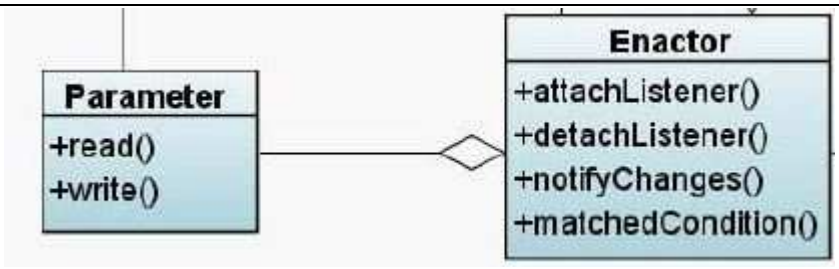



Kuva 5. Tarjotun ja vaaditun rajapinnan kuvaaminen UML:ssä. [OMG, 2006]




Kuva 6. Vaihtoehtoinen tapa kuvata rajapinta. [Vromant et al., 2011]

Luokkien väliset suhteet kuvataan viivoilla, jotka piirretään luokkia kuvaavien suorakulmioiden välille. Luokkien välisiä suhteita on kolmea eri tyyppiä, assosiaatiot, riippuvuussuhteet ja periytymissuhteet. Assosiaatioita on eri tyyppejä: tavallinen assosiaatio, refleksiivinen assosiaatio, koosteassosiaatio ja aito kooste. Tavallinen assosiaatio merkitään viivalla luokkasymbolien välillä. Assosiaatiot voidaan nimetä ja nimen yhteyteen voidaan merkitä nuoli kuvaamaan lukusuuntaa. Assosiaation päitä kutsutaan rooleiksi ja ne voidaan myös nimetä. Assosiaation yhteydessä kuvataan myös kertautuminen, mikä tarkoittaa sitä, kuinka monta assosiaation toisen pään luokan ilmentymää liittyy assosiaation toisen pään luokan ilmentymään. Merkki * ilmaisee, että toisen pään luokan ilmentymiä voi olla kuinka monta tahansa. Refleksiivinen assosiaatio tarkoittaa, että assosiaatio päättyy samaan luokkaan, josta se lähtee. Koosteassosiaatiolla ilmaistaan, että jonkin luokan ilmentymä kuuluu johonkin toiseen luokkaan tai on osa sitä. Se merkitään vinoneliöllä assosiaatiota kuvaavan viivan siinä päässä, jonka luokan ilmentymän osa toisen luokan ilmentymä on. Kuvassa 7 on osa Enactor-suunnittelumallin luokkakaaviota, jossa Enactor- ja Parameter-luokkien välillä on koosteassosiaatio. [OMG, 2006]




Kuva 7. Esimerkki koosteassosiaatiosta. [Riva et al., 2006]

Aito kooste tarkoittaa, että toisen luokan ilmentymä ei voi olla olemassa ilman toisen luokan ilmentymää ja se merkitään mustalla vinoneliöllä. Riippuvuussuhteella kuvataan luokkien väliset väliaikaiset suhteet, kuten olioiden luominen tai toisen luokan palveluiden käyttäminen, ja se esitetään nuolella, jonka varsi on katkoviiva ja kärki väkäpäinen. Nuoli osoittaa siihen luokkaan, jonka palveluja käytetään, kuten kuvassa 5, jossa TheftAlarm-luokka käyttää ISensor-rajapinnan palvelui-

ta. Periytymissuhteet kuvaavat erikoistumista ja yleistystä yli- ja aliluokkien välillä. Ne esitetään luokkakaavioissa nuolena, jonka kärki on tasasivuinen kolmio ja joka osoittaa yliluokkaan. [OMG, 2006]

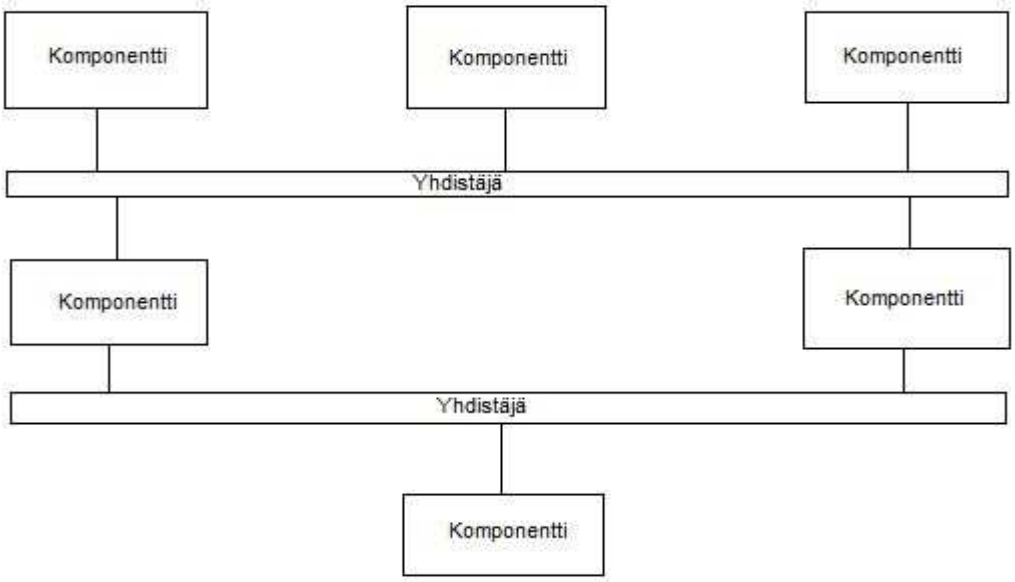
# 4. Arkkitehtuurityylit

Tässä luvussa käsittelen eri arkkitehtuurityylejä. Arkkitehtuurityylillä tarkoitetaan yleistä mallia, jonka mukaan järjestelmä organisoidaan ylimmällä abstraktiotasolla ja joka määrää järjestelmän teknisen luonteen. Yleisesti tunnettuja arkkitehtuurityylejä ovat kerrosarkkitehtuurit, viestinvälitysarkkitehtuurit, tietovuoarkkitehtuurit, asiakas-palvelin-arkkitehtuurit, MVC-arkkitehtuurit ja tulkkiarkkitehtuurit.

Oreizy ja muut [2008] käsittelevät arkkitehtuurityylejä, jotka tukevat itseohjautuvuutta. Tällaisia tyylejä ovat Pipe and Filter, Weaves, Publish-Subscribe, C2, Tile Style, Representational State Transfer (REST), Computational REST (CREST) ja MapReduce. Tässä luvussa esittelen näitä arkkitehtuurityylejä yleisellä tasolla. Lisäksi esittelen kaksi kehystä, SASSY:n ja Rainbow:n, joita voidaan käyttää itseohjautuvien järjestelmien suunnittelussa. Ne eivät varsinaisesti ole arkkitehtuurityylejä, sillä Rainbow soveltuu käytettäväksi useimpien arkkitehtuurityylien kanssa ja SASSY soveltuu palveluperustaisiin arkkitehtuureihin, mutta olen ottanut ne mukaan tähän tutkielmaan, koska ne tukevat useita eri itseohjautuvuuteen liittyviä ominaisuuksia.

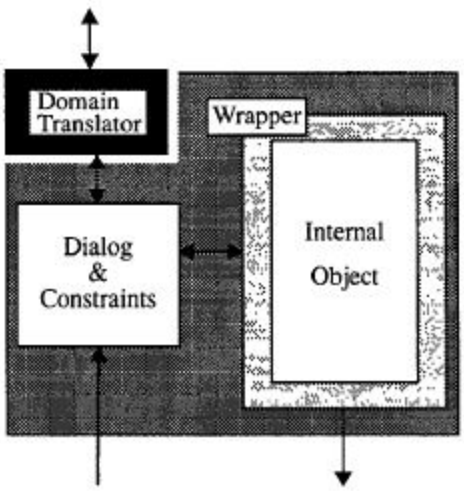
## 4.1. C2-arkkitehtuuri

C2-arkkitehtuuri on eräänlainen yhdistelmä viestinvälitysarkkitehtuurista ja kerrosarkkitehtuurista. Se koostuu komponenteista ja yhdistäjistä, joilla on kaksi porttia, yksi yläportti ja yksi alaportti, joiden kautta komponentit lähettävät viestejä toisilleen. Komponentin yläportti voidaan liittää yhdistäjän alaporttiin ja komponentin alaportti yhdistäjän yläporttiin. Yhteen yhdistäjään voidaan liittää useita komponentteja ja toisia yhdistäjiä. Kuvassa 8 on yksinkertainen C2-arkkitehtuuri.




Kuva 8. C2-arkkitehtuuri. [Taylor et al., 1995]

Komponenttien sisäinen rakenne on esitetty kuvassa 9. Komponentilla on sisäinen olio, joka tarjoaa rajapinnan. Olion ympärillä on ns. kääre. Kääre toimii siten, että kun jotakin olion rajapinnan metodia kutsutaan, kääre muodostaa kutsusta ja sen paluuarvosta ilmoituksen, jonka se välittää komponentin alaporttiin liitetylle yhdistäjälle. Lisäksi komponentti sisältää ohjelman, joka ylläpitää vuoropuhelua ja rajoitteita. Komponentti voi sisältää myös osakomponentin, jonka tehtävänä on kääntää komponentin yläporttiin liitetyltä yhdistäjältä tulevat viestin komponentin ymmärtämään muotoon. [Taylor et al., 1995]




Kuva 9. Komponentin rakenne C2-arkkitehtuurissa. [Taylor et al., 1995].

Komponentit voivat lähettää kahdenlaisia viestejä, ilmoituksia ja pyyntöjä. Ilmoitukset kulkevat arkkitehtuurissa alaspäin ja pyynnöt ylöspäin. Ilmoituksilla komponentit lähettävät toisilleen tiedon niiden sisäisten olioiden tiloissa tapahtuneista muutoksista. Pyyntöjen avulla komponentit sen sijaan ohjeistavat yläpuolellaan olevia komponentteja toimimaan halutulla tavalla. Ilmoituksilla viestitään siis sitä, mitä toimenpiteitä on tehty, millä parametreilla ja mitä paluuarvoja on saatu, ja pyynnöillä viestitään sitä, mitä toimenpiteitä halutaan tehdä. [Taylor et al., 1995]

Yhdistäjien tehtävänä on liittää komponentit toisiinsa sekä reitittää ja välittää viestejä. Ilmoitukset voidaan välittää joko kaikille yhdistäjän alaporttiin liitetyille komponenteille ja pyynnöt vastaavasti kaikille yläporttiin liitetyille komponenteille, tai ne voidaan välittää vain sellaisille komponenteille, jotka ovat tilanneet kyseisentyyppiset viestit. Viestejä voidaan välittää myös siten, että ne lähetetään komponenteille tietyssä järjestyksessä niin kauan kunnes jokin määrätty ehto täyttyy. [Taylor et al., 1995]

### 4.2. PitM-arkkitehtuuri

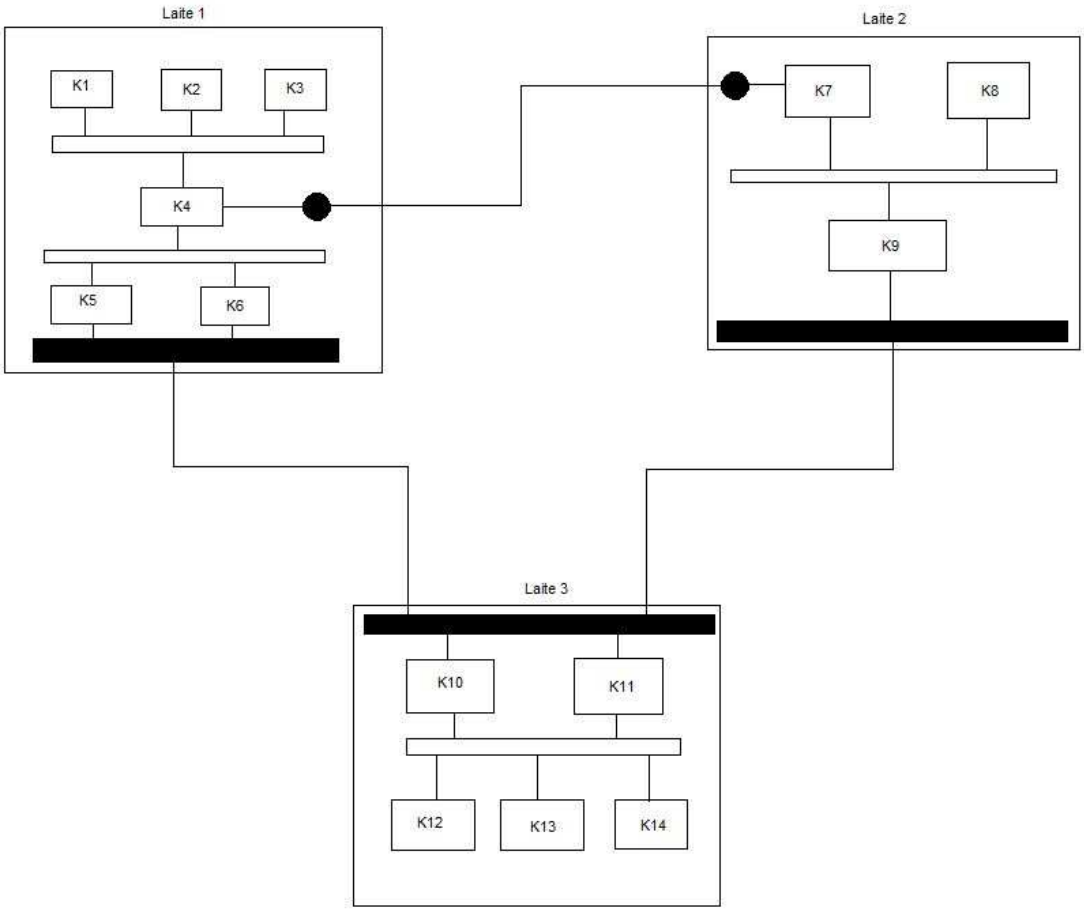
PitM-arkkitehtuuri perustuu C2-arkkitehtuuriin. Se koostuu komponenteista ja yhdistäjistä, jotka kommunikoivat välittämällä viestejä. C2-arkkitehtuurin ylä- ja alaporttien lisäksi komponenteilla voi olla myös sivuportteja. Ilmoitusten ja pyyntöjen lisäksi PitM-tyylissä on myös kolmas viestityyppi, *vertaisviestit* (peer). Lisäksi PitM-arkkitehtuurissa voi olla myös ns. *reunayhdistäjiä* (border

connector), jotka toimivat eri laitteissa sijaitsevien komponenttien välissä. [Medvidovic and Mikic-Rakic, 2001]

Komponenttien sivuportteihin liitetään ns. *vertaisyhdistäjiä* (peer connector), joiden kautta komponentit voivat lähettää toisilleen vertaisviestejä. Kun viesti tulee vertaisyhdistäjään, se välitetään eteenpäin muiden porttien kautta niille komponenteille, jotka on liitetty portteihin. Viestit voidaan välittää myös määrättyjen ehtojen mukaisesti, samoin kuin C2-tyylissä. [Medvidovic and Mikic-Rakic, 2001]

PitM-arkkitehtuurissa komponentit eivät voi lähettää toisilleen vertaisviestejä, mikäli niiden välillä on vertikaalinen yhteys. Myöskään vertaisyhdistäjien ja tavallisten yhdistäjien väliset viestit eivät ole sallittuja, sillä silloin vertaisviestit pitäisi muuttaa pyynnöiksi ja ilmoituksiksi ja päinvastoin. [Medvidovic and Mikic-Rakic, 2001]

Kuvassa 10 on esitetty PitM-arkkitehtuurin rakenne järjestelmässä, jossa on kolme laitetta. Mustat suorakaiteet kuvaavat reunayhdistäjiä ja mustat ympyrät vertaisyhdistäjiä. Laitteet 1 ja 2 ovat laitteen 3 yläpuolella, joten vertaisyhdistäjät niiden komponenttien ja laitteen 3 komponenttien välillä eivät ole sallittuja. Laitteiden 1 ja 2 välillä sen sijaan ei ole vertikaalista yhteyttä, joten niiden sisältämät komponentit voivat lähettää vertaisviestejä toisilleen vertaisyhdistäjien kautta.




Kuva 10. PitM-arkkitehtuuri. [Medvidovic and Mikic-Rakic, 2001]

### 4.3. Weaves-arkkitehtuuri

Pipe and Filter -tyylissä olennaista on erilliset komponentit sekä komponenttien väliset yhdistäjät, rajapintojen käyttö ja standardimuotoiset viestit. Arkkitehtuuri koostuu *suodatinkomponenteista* (filter), jotka saavat sisäänsä tietoa, käsittelevät sen, ja siirtävät *väylää* (pipe) pitkin eteenpäin seuraavalle suodatinkomponentille. Uuden järjestelmän luominen olemassa olevista suodattimista ja väylistä on yksinkertaista, mutta tyyli ei kuitenkaan tue ajonaikaista muutosta. Esimerkiksi ajonaikainen uudelleenreititys yhdestä komponentista toiseen ei onnistu. [Oreizy et al., 2008]

Weaves-tyyli on kehitetty Pipe and Filter -tyylin pohjalta ja se mahdollistaa ajonaikaiset muutokset yhdistämällä viestien puskuroinnin ja tietovoiden hallintamekanismit. Kun väylä irrotetaan sen jälkeisestä suodattimesta, väylän puskuri täyttyy viesteistä. Kun puskurin kapasiteetti on melkein täynnä, väylä pyytää sitä edeltävän suodattimen rajapintaa hidastamaan viestien tuotantoa siihen asti kunnes uuden komponentin liittäminen on suoritettu loppuun ja puskuri pystyy taas käsittelemään uusia viestejä. [Oreizy et al., 2008]

Weaves-arkkitehtuuri on verkosto, joka koostuu samanaikaisesti toimivista komponenteista, jotka kommunikoivat keskenään lähettämällä olioita. Weaves-arkkitehtuurityyli on kehitetty järjestelmille, jotka käsittelevät tietovirtoja, mutta se poikkeaa muista tietovuoarkkitehtuureista siten, että se keskittyy välineisiin, sitä voidaan tarkkailla jatkuvasti ja sen komponentit voidaan järjestää uudelleen dynaamisesti. [Gorlick and Razouk, 1991]

Weaves-tyyli sopii järjestelmiin, jotka ovat poikkeuksellisen laajoja, satojentuhansien koodirivien laajuisia, joiden täytyy käsitellä tietoja reaaliaikaisesti tai ainakin lähes reaaliaikaisesti, jotka ovat vaikeaselkoisia ja jotka muuttuvat jatkuvasti. Weaves ratkaisee nämä vaatimukset mahdollistamalla järjestelmän jatkuvan inkrementaalisen kehittymisen, jatkuvan tarkkailemisen ja havainnoinnin komponenttien yhtäaikaisen kommunikaation sekä tarjoamalla mahdollisuuksia rinnakkaisuuteen ja jatkuvaan skaalautumiseen. [Gorlick and Razouk, 1991]

Weaves-arkkitehtuurissa verkoston komponenteilla on yksi tietty tarkasti määritelty tehtävä, jota ne suorittavat. Ne saavat syötteenä olioita ja tuottavat olioita. Jokaisella komponentilla on oma säikeensä. Olioita siirretään komponentista toiseen porttien välityksellä. Portit yhdistävät komponentit jonoihin, joiden tehtävänä on puskuroida ja synkronoida komponenttien välisiä viestejä. Oliot, portit ja jonot ovat passiivisia, ja ainoastaan komponentit ovat aktiivisia. Ne vastaanottavat olioita porteilta, kutsuvat olioiden tarjoamia metodeja ja lähettävät olioita eteenpäin. [Gorlick and Razouk, 1991]

Portit ja jonot ovat tyyppiriippumattomia kaksikerroksisia tiedonsiirtopalveluita. Niitä käyttävät komponentit eivät tiedä vastaanottamansa olion alkuperää tai lähettämänsä olion määränpäätä. Portit kapseloivat oliot siten, että niiden tyyppiä ei tiedetä, ja purkavat kapseloinnin. Jonot puskuroivat kapseloituja olioita. Kapselointi ja tyyppiriippumattomuus lisäävät yhdistettävyyttä. Komponentteja, portteja ja jonoja voidaan siten liittää toisiinsa joustavammin. Porttien tehtävänä on

myös huolehtia olioiden siirtämisen atomisuudesta, siis siitä, että olio voi olla vain yhdessä paikassa kerrallaan, joko komponentin sisällä tai jonossa. Portti hoitaa tämän siten, että kun komponentti lähettää sille olion, se varmistaa sen toimittamisen jonoon. Samoin oliot pysyvät jonossa niin kauan kunnes portti pystyy varmistamaan sen toimituksen eteenpäin. Atomisuus mahdollistaa sen, että verkostot voidaan dynaamisesti järjestää uudelleen siten, että olioiden muodostamat tietovuot eivät katkea. Oliot jäävät odottamaan portteihin, jos niitä ei heti voida lähettää vastaanottavalle komponentille tai jonolle. Jos jono täyttyy, se ei voi enää vastaanottaa uusia olioita, vaan jonoon kytketty portti lähettää virheilmoituksen olion lähettäneelle komponentille. [Gorlick and Razouk, 1991]

Komponentit ovat verkostojen aktiivisia toimijoita. Niiden voidaan ajatella olevan eräänlaisia pieniä työkaluja, joilla jokaisella on tietty tehtävä. Ne koostuvatkin kyseisestä rutiinista, jota ne suorittavat, argumentteja sisältävästä vektorista sekä säikeestä, joka suorittaa komponentin tehtävää. Rutiini on komponentin toiminnallisuus, joka käsittelee syötteenä tulevia olioita ja tuottaa eteenpäin lähetettäviä olioita. Vektori on jaettu kolmeen osavektoriin. Ensimmäiseen osavektoriin kuuluvat ne argumentit, jotka eivät ole portteja, toiseen osavektoriin kuuluvat ne portit, joista komponentti lukee syötteenä tulevia olioita, ja kolmanteen osavektoriin kuuluvat ne portit, joihin komponentti lähettää tuottamansa oliot. Säie liittyy komponenttiin koko sen elinkaaren ajan. [Gorlick and Razouk, 1991]

Kaikki komponentit toteuttavat tietyt ohjausmetodit. Start-metodi käynnistää komponentin suorituksen. Suspend-metodi pysäyttää sen siten, että suoritusta voidaan myöhemmin jatkaa samasta kohtaa, johon se pysäytettiin. Resume-metodi keskeyttää suorituksen, jolloin komponentti palaa samaan tilaan, jossa se oli ennen komponentin käynnistystä. Sleep-metodi keskeyttää suorituksen määrätyksi ajaksi, jonka jälkeen se jatkuu normaalisti. Abort-metodi keskeyttää suorituksen, mutta sen jälkeen sitä ei voida jatkaa samasta kohdasta. [Gorlick and Razouk, 1991]

Komponentit voidaan jakaa kahteen eri tyyppiin. Ensimmäistä tyyppiä olevat metodit tukevat ulkopuolisten rutiinien sisällyttämistä olemassa oleviin verkostoihin. Tällöin rutiini eristetään kaikista porttien kautta lähetettäviin ja vastaanotettaviin olioihin liittyvistä yksityiskohdista. Komponenttia luotaessa määritellään komponentin argumenttivektorin ja rutiinin parametrien väliset yhteydet. Kun uusi komponentti liitetään verkostoon, odotetaan, että olioita saapuu syötettä tuottaviin portteihin. Kun kaikki oliot on vastaanotettu, rutiinia kutsutaan parametreilla, jotka saadaan selville argumenttivektorin avulla. Toinen tyyppi ovat komponentit, jotka on alusta lähtien suunniteltu ja toteutettu Weaves-arkkitehtuurin komponenteiksi. Niissä rutiinia ei mitenkään eristetä verkostosta, vaan rutiinia kutsutaan ainoastaan kerran, kun komponentti käynnistetään. Rutiinin vastuulla on purkaa argumenttivektori, kerätä oliot syötettä tuottavilta porteilta, käsitellä saamansa oliot ja lähettää ne eteenpäin. [Gorlick and Razouk, 1991]

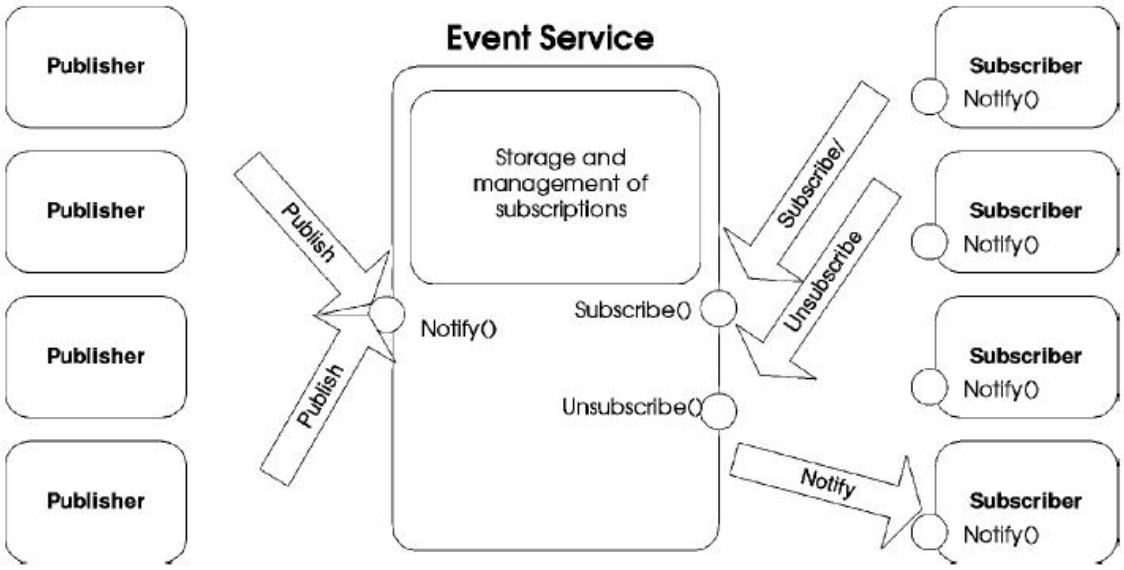
Koska Weaves-arkkitehtuurityylissä komponenttien väliset viestit ovat olioita, joiden tyyppejä niitä käsittelevät komponentit eivät tiedä, komponentteja on helppo poistaa ja lisätä. Näin ollen Weaves-arkkitehtuurityyliä voidaan käyttää tai siitä voidaan ottaa vaikutteita sellaisten järjestelmien suunnitteluun, joissa halutaan mahdollistaa uudelleenkonfigurointi. Koska Weaves perustuu tie-

tovuoarkkitehtuuriin, se tukee erinomaisesti itseoptimoituvuutta. Jos jonoihin kerääntyy paljon olioita, voidaan järjestelmään liittää uusia portteja, jonoja tai komponentteja, jolloin oliot voidaan jakaa useille samaa tehtävää tekevälle komponentille. Tällöin järjestelmän suorituskyky saadaan pidettyä toivotulla tasolla, eikä yksi komponentti muodostu pullonkaulaksi.

### 4.4. Publish-Subscribe

Publish-subscribe -arkkitehtuuri on viestinvälitysarkkitehtuuri, jossa tuottajan tehtävänä on lähettää viestejä ja tilaajan tehtävänä on vastaanottaa niitä. Tilaaja rekisteröityy vastaanottajaksi ja tuottaja pitää kirjaa vastaanottajista. [Oreizy et al., 2008]

Publish-subscribe -arkkitehtuurissa tilaajat ilmaisevat olevansa kiinnostuneita tietynlaisista tapahtumista. Kun tuottaja luo sellaisen tapahtuman, joka vastaa tilaajan kiinnostusta, se ilmoittaa tapahtumasta tilaajalle. Tuottajat julkaisevat tietoa tapahtumapalveluun ja tilaajat rekisteröityvät tilaamaan haluamaansa tietoa palvelusta. Tietoa kutsutaan siis *tapahtumaksi* (event), ja tiedon välittämisestä käytetään termiä *ilmoitus* (notification). Kuvassa 11 esitetään Publish-subscribe -arkkitehtuurin toimintaperiaate.




Kuva 11. Publish-subscribe -arkkitehtuuri. [Eugster et al., 2003]

Tapahtumapalvelu toimii välittäjänä tilaajien ja tuottajien välissä. Tilaajat rekisteröityvät ilmoituksien vastaanottajiksi kutsumalla Subscribe-operaatiota. Tilauksen voi lopettaa kutsumalla Unsubscribe-operaatiota. Tuottajat kutsuvat Publish-operaatiota luodakseen tapahtumaan. Tapahtumapalvelu välittää tapahtuman kaikille tilaajille, jotka haluavat vastaanottaa kyseisenlaisia tapahtumia. Tuottajat voivat myös mainostaa tulevia tapahtumiaan advertise-operaatiolla. Tämä tukee itseohjautuvuutta, sillä tapahtumapalvelu voi varautua mahdollisiin tuleviin tilauksiin, kun tilaajat saavat tietää, millaisia uusia tapahtumia ne jatkossa voivat tilata. [Eugster et al., 2003]

Koska tapahtumapalvelu toimii välittäjänä tilaajien ja tuottajien välillä, ne eivät ole tietoisia toisistaan. Tuottaja ei siis tiedä, mitkä komponentit ovat kiinnostuneita sen luomista tapahtumista, eikä tilaaja tiedä, mikä komponentti tuottaa sen tilaamaa tietoa. Välittäjä mahdollistaa myös sen, että tilaajien ja tuottajien ei välttämättä tarvitse toimia samanaikaisesti. Tuottaja voi luoda tapahtuman, kun tilaaja ei ole aktiivinen, ja tilaaja voi rekisteröityä jonkin tiedon vastaanottajaksi, kun tapahtuman alkuperäinen tuottaja ei ole aktiivinen. Lisäksi tuottajia ei estetä niiden tuottaessa tapahtumia ja tilaajille voidaan ilmoittaa takaisinkutsun avulla asynkronisesti tapahtumasta, jos ne suorittavat jotakin rinnakkaista toimenpidettä. Tapahtumien tuottaminen ja tilausten vastaanottaminen eivät tapahdu komponenttien pääsäikeessä, joten ne eivät siksi ole synkronisia. Tilaajien ja tuottajien erottaminen toisistaan lisää järjestelmän skaalautuvuutta, koska se poistaa riippuvuudet toimijoiden väliltä, minkä vuoksi se sopii hyvin hajautettuihin järjestelmiin. [Eugster et al., 2003]

Koska tilaajat ovat kiinnostuneita tietynlaisista tapahtumista tai tapahtumamalleista kaikkien tapahtumien sijaan, on syntynyt erilaisia variaatioita Publish-subscribe -mallista. Tilaajat voivat rekisteröityä vastaanottamaan tapahtumia joko aiheen, sisällön tai tyypin perusteella. [Eugster et al., 2003]

Aiheperustaisessa Publish-subscribe -mallissa välittäjäkomponentti tarjoaa mahdollisuuden luonnehtia ja luokitella tapahtumien sisältöä. Tuottajat voivat julkaista tapahtumia, joilla on tunnisteena jokin avainsana, ja tilaajat voivat rekisteröityä sellaisten tapahtumien vastaanottajiksi, joita on luonnehdittu jollakin tietyllä avainsanalla. Käytännössä jokaisen yksittäisen aiheen tilaajat ja tuottajat muodostavat erillisen viestintäkanavan. Jokainen aihe voidaan siis ajatella omana tapahtumapalvelunaan. [Eugster et al., 2003]

Sisältöperustaisessa Publish-subscribe -mallissa tapahtumia ei luokitella ennalta määriteltyjen ulkoisten kriteerien perusteella, vaan tapahtumien ominaisuuksien perusteella. Ominaisuudet voidaan kuvata esimerkiksi tapahtumiin liitettävässä metatiedossa. Tilaajat rekisteröityvät vastaanottamaan tapahtumia määrittelemällä suodattimia. Suodattimet tunnistavat halutut tapahtumat niissä määriteltyjen tapahtumien ominaisuuksia koskevien rajoitteiden avulla. [Eugster et al., 2003]

Tyyppiperustaisessa Publish-subscribe -mallissa luokitellaan tapahtumat niiden tyypin mukaan, sillä tapahtumat voivat olla rakenteeltaan samankaltaisia. Se mahdollistaa kielen ja väliohjelmiston lähemmän integroinnin. Tapahtumatyypeillä voi olla alityyppejä samaan tapaan kuin ohjelmointikielissä voidaan periyttää luokista aliluokkia. Alityyppien ja ylityyppien avulla tilaajat voivat rekisteröityä vastaanottamaan erityyppisiä tapahtumia samalla tilauksella. [Eugster et al., 2003]

Publish-subscribe -mallin avulla voidaan luoda järjestelmiin tiedon keräämistä ja tarkkailemista tukevaa toiminnallisuutta. Järjestelmän komponentit voivat esimerkiksi tuottaa tapahtumia, joissa kuvataan järjestelmän suorituskykyä, ja tarkkailijakomponentti rekisteröityy tilaamaan sellaisia tapahtumia. Mikäli järjestelmän ympäristö vaikuttaa sen toimintaan, se voi kerätä tietoa ympäristöstä sensoreiden avulla. Tällöin sensorit toimivat tuottajina, jotka julkaisevat järjestelmän käyttöön ympäristön tilaa kuvaavia tapahtumia.

### 4.5. REST ja CREST

REST-tyylin tunnetuin toteutus lienee Internet. Se muuttuu jatkuvasti palvelimien, asiakkaiden, porttien ja välittäjien vaihtuessa. Se on suunniteltu hypermediadokumenttien jakamiseen. REST:ssä tiedon siirtäminen perustuu resursseihin, joihin viitataan URL:illa. Resurssit esitetään tavuina sekä tavuja kuvaavana metatietona. CREST sen sijaan perustuu tiedon siirtämisen sijaan tiedon käsittelyn siirtämiseen. Tiedon käsittely voi olla esimerkiksi tekstin käsittelyä tai kuvan muokkaamista. CREST:ssä resurssi on ohjelma sekä sitä kuvaava metatieto. [Oreizy et al., 2008]

Representational State Transfer (REST) on arkkitehtuurityyli, joka on suunniteltu hypermediadokumenttien jakamiseen. Siinä on kolmenlaisia arkkitehtuurielementtejä: prosessointielementtejä eli komponentteja, tietoelementtejä ja yhdistämiselementtejä eli yhdistäjäkomponentteja. REST ei ota huomioon komponenttien toteutuksen tai protokollasyntaksin yksityiskohtia, vaan keskittyy komponenttien rooleihin, komponenttien välisen vuorovaikutuksen rajoitteisiin ja siihen, miten komponentit tulkitsevat tietoelementtejä. [Fielding and Taylor, 2000]

Kun linkkiä klikataan, tieto täytyy siirtää varastointipaikasta käyttöpaikkaan. Tämä voidaan tehdä kolmella eri tavalla. Ensinnäkin informaatio voidaan muuttaa esitettävään muotoon samassa paikassa, jossa sitä säilytetään, ja lähettää se siinä muodossa vastaanottajalle. Toiseksi, se voidaan kapseloida sen käsittelevän prosessorin kanssa ja lähettää ne vastaanottajalle yhdessä. Kolmas vaihtoehto on lähettää tieto vastaanottajalle sellaisenaan ja liittää mukaan metatieto, joka kuvaa tiedon tyypin, jolloin vastaanottaja voi valita itse, miten tietoa käsitellään. REST tarjoaa mahdollisuuden yhdistää nämä vaihtoehdot, sillä se keskittyy jaettuun ymmärrykseen tietotyypeistä, joihin liittyy metatietoa, mutta johon pääsee käsiksi vain standardisoidun rajapinnan kautta. REST:n komponentit kommunikoivat keskenään siirtämällä tiedon kuvauksia (representation) formaatissa, joka on jonkin standarditietotyypin mukainen. Tämä tietotyyppi valitaan vastaanottajan ja tiedon luonteen mukaan. Tällöin rajapinta piilottaa tiedon siitä, onko tiedon kuvaus samassa muodossa kuin sen lähde tai onko se johdettu suoraan lähteestä. [Fielding and Taylor, 2000]

REST:ssä tiedon abstraktiota kutsutaan resurssiksi. Resurssi voi olla mikä tahansa tieto, joka voidaan nimetä jollakin tavalla, esimerkiksi dokumentti, kuva tai kokoelma muita resursseja. Resurssilla voi olla eri tila eri ajanhetkinä. Tiettyyn resurssiin, joka osallistuu komponenttien väliseen vuorovaikutukseen, viitataan *resurssin tunnisteella* (resource identifier). Yhdistäjäkomponentit tarjoavat rajapinnan, jonka kautta resurssin arvojoukkoa voidaan käsitellä. [Fielding and Taylor, 2000]

REST:n komponentit suorittavat toimenpiteitä resursseille käyttämällä sen resurssin senhetkistä kuvausta ja siirtämällä tätä kuvausta komponentilta toiselle. Tiedon kuvaus on tavujen sekvenssi ja metatieto, joka kuvaa tavuja. Metatieto koostuu avain-arvo -pareista, jossa avain vastaa standardia, joka määrittelee arvon rakenteen ja semantiikan. Vastausviestit voivat sisältää sekä kuvauksen metatietoa että resurssin metatietoa. [Fielding and Taylor, 2000]

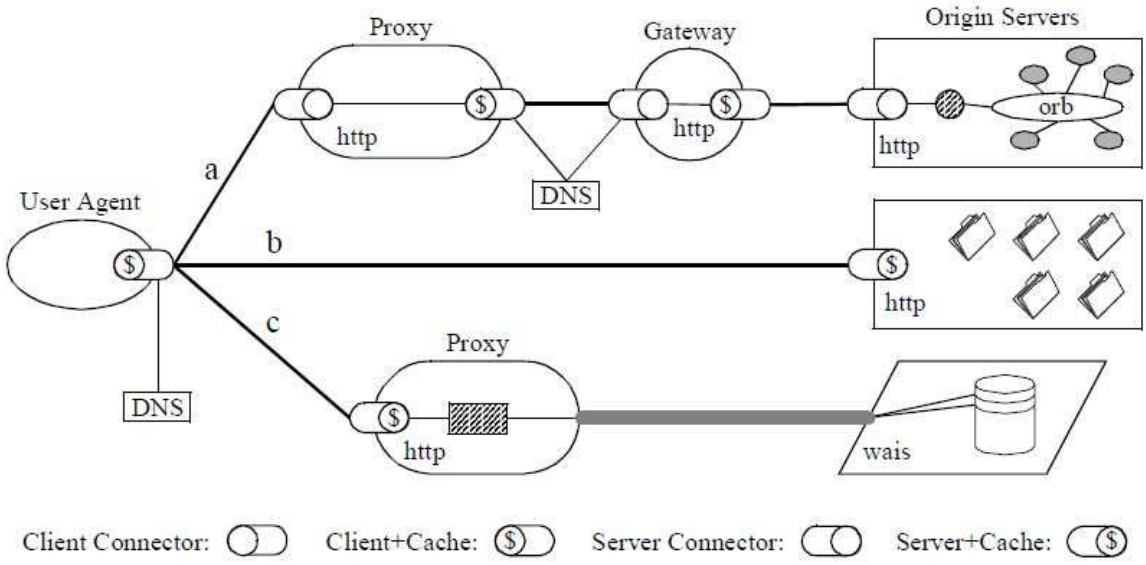
Kontrollitieto on tietoa, joka määrittelee komponenttien välisen viestin tarkoituksen. Sitä voidaan käyttää kertomaan esimerkiksi, että viesti on pyyntö suorittaa jokin toimenpide tai se voi ker-

toa vastauksen merkityksen. Kontrollitietoa voidaan käyttää myös parametrisoimaan pyyntöjä tai korvaamaan joidenkin yhdistävien elementtien oletuskäyttäytyminen. [Fielding and Taylor, 2000]

Tiedon kuvauksen tietoformaattia sanotaan mediatyypiksi. Kuvaus voidaan sisällyttää viestiin ja vastaanottaja voi käsitellä sen viestin kontrollitiedon ja mediatyypin mukaan. Jotkut mediatyypit käsitellään automaattisesti, jotkut muunnetaan esitettäväksi käyttäjälle ja jotkut voidaan käsitellä molemmilla tavoilla. Yhdistettyjen mediatyyppien avulla voidaan sisällyttää useita tiedon kuvauksia yhteen viestiin. [Fielding and Taylor, 2000]

REST-tyylissä on viisi erilaista yhdistäjätyyppiä, *asiakas* (client), *palvelin* (server), *välimuisti* (cache), *resolveri* (resolver) ja *tunneli* (tunnel). Yhdistäjät edustavat abstrakteja rajapintoja, joiden kautta komponentit kommunikoivat. Koska komponenttia voidaan käyttää vain tällaisen rajapinnan kautta, rajapinnan toteuttava komponentti voidaan helposti korvata toisella saman rajapinnan toteuttavalla komponentilla ilman, että se vaikuttaa käyttäjään. Pääasialliset yhdistäjätyypit ovat asiakas- ja palvelinyhdistäjä. Asiakasyhdistäjän tehtävänä on aloittaa kommunikaatio tekemällä pyyntö. Palvelinyhdistäjä kuuntelee yhteyksiä ja vastaa pyyntöihin. Samalla komponentilla voi olla sekä asiakas- että palvelinyhdistäjiä. Välimuistiyhdistäjä voi olla asiakas- tai palvelinyhdistäjän rajapinnassa. Sen tehtävä on tallentaa vastauksia välimuistiin, jotta niitä voidaan myöhemmin käyttää uudestaan. Asiakas voi käyttää välimuistia välttääkseen toistuvaa kommunikaatiota, ja palvelin välttääkseen toistuvaa saman vastauksen generoimista, mikä vähentää latenssia. Resolveri muuntaa resurssien tunnisteita osoitetiedoiksi, joita tarvitaan komponenttien välisten yhteyksien muodostamiseen. Tunnelin tehtävänä on siirtää kommunikaatio yhteyksien välisten rajojen, esimerkiksi palomuurin, yli. [Fielding and Taylor, 2000]

Komponenttityyppejä on neljä, *alkuperäpalvelin* (origin server), *portti* (gateway), *välityspalvelin* (proxy) ja *käyttäjä* (user agent). Alkuperäpalvelin käyttää palvelinyhdistäjää hallitakseen pyydetyn resurssin nimiavaruutta. Se määrittää resurssiensa esitystavat ja sen täytyy olla niiden pyyntöjen lopullinen vastaanottaja, joiden tarkoituksena on muokata sen resurssien arvoja. Jokainen alkuperäpalvelin tarjoaa rajapinnan palveluihinsa. Käyttäjäkomponentti käyttää asiakasyhdistäjää alustaakseen pyynnön ja on vastauksen lopullinen vastaanottaja. Välityspalvelin ja portti ovat välittäjäkomponentteja. Välittäjäkomponentit voivat toimia sekä asiakkaana että palvelimena välittääkseen pyyntöjä ja vastauksia. Välityspalvelimen tehtävänä on tarjota asiakkaalle rajapinta muiden palveluiden kapseloimiseen, tiedon muuntamiseen, suorituskyvyn parantamiseen tai tietoturvan takaamiseen. Portti on välittäjäkomponentti, jota verkko tai alkuperäpalvelin käyttää samoihin tarkoituksiin. Kuvassa 12 on esimerkki REST-arkkitehtuurista.




Kuva 12. REST-arkkitehtuuri. [Fielding and Taylor, 2000]

CREST eroaa REST:stä siten, että resurssi on tiedon käsittelyn abstraktio. Siihen viitataan URL:lla samoin kuin perinteisessä REST:ssäkin. Resurssin tilan esitystapa voi olla esimerkiksi ohjelma, joka käsittelee tietoa. CREST:ssä kaikki tiedon käsittely on kontekstitonta. Se tarkoittaa, että kaikki vuorovaikutus itsessään sisältää kaiken tarvittavan tiedon pyynnön ymmärtämiseksi. [Oreizy et al., 2008]

CREST:ssä on viisi pääperiaatetta, joiden mukaan siihen perustuva järjestelmä suunnitellaan. Ensinnäkin, tiedon käsittelyn abstraktio on resurssi, johon viitataan URL:lla. Mikä tahansa tietoa käsittelevä toiminto voi olla resurssi, esimerkiksi kuvan muokkaaminen tai tekstin käsittely. Toiseksi, resurssin esitystapa on ohjelma, sulkeuma, jatkumo tai sidosympäristö sekä resurssiin liitetty sitä kuvaava metatieto. Tästä syystä CREST sisältää välikerroksen abstraktin kerroksen ja konkreettisen esitystavan välillä. Kolmanneksi, kaikki tiedon käsittely tapahtuu kontekstittomasti. Se tarkoittaa sitä, että kaikki vuorovaikutus sisältää kaiken tiedon, jota tarvitaan pyynnön ymmärtämiseksi, riippumatta sitä edeltävistä pyynnöistä. Edeltäviä esitystapoja voidaan käyttää tilojen siirtämiseen tiedonkäsittelyjen välillä. Neljänneksi, CREST:ssä on vain muutama primitiivinen operaatio, mutta lisäksi voidaan käyttää resurssikohtaisia operaatioita. Viides periaate on, että välittäjien käyttö on suositeltavaa. Suodatin- ja uudelleenohjausvälittäjät voivat käyttää sekä metatietoa että tiedonkäsittelyä pyynnöissä ja vastauksissa lisätäkseen, rajoittaakseen tai muuttaakseen pyyntöjä ja vastauksia siten, että sekä asiakas että alkuperäpalvelin ovat tietoisia siitä. [Erenkrantz et al., 2007]

### 4.6. MapReduce

MapReduce ei varsinaisesti ole arkkitehtuurityyli, vaan pikemminkin ohjelmointimalli, jota voidaan käyttää suurten tietomäärien prosessoimiseen. Siinä määritellään kaksi toimintoa, joista toinen jakaa tietomäärän pienempiin osiin, jotka sitten käsitellään rinnakkaisesti, ja toinen sulauttaa käsitel-

lyn tiedon takaisin yhteen. Mikäli joku tietoa käsittelevistä yksiköistä kaatuu, sen käsittelemä tieto ohjataan automaattisesti jollekin toimivalle yksikölle. [Oreizy et al., 2008]

MapReduce soveltuu käytettäväksi järjestelmissä, joissa täytyy käsitellä suuria määriä tietoa. Sitä on käytetty esimerkiksi tiedonlouhintaan, tekstinkäsittelyyn, koneoppimiseen ja tilastolliseen konekäännökseen liittyvissä järjestelmissä. MapReduce perustuu tiedonkäsittelyn jakamiseen kahteen perustoimintoon, Map- ja Reduce-funktioihin. Järjestelmä ottaa syötteen joukkona avain-arvo -pareja ja myös tuottaa joukon avain-arvo -pareja. Map-funktio ottaa yhden syöteparin ja tuottaa joukon keskiasteen avain-arvo -pareja. MapReduce-kirjasto kokoaa yhteen kaikki keskiasteen arvot, jotka liittyvät samaan avaimeen, ja välittää ne Reduce-funktiolle. Reduce-funktio ottaa syötteenä avaimen ja joukon siihen liittyviä arvoja. Arvot sulautetaan yhteen, jotta muodostuu mahdollisesti pienempi joukko arvoja. Yleensä yksi Reduce-funktion kutsu tuottaa ainoastaan yhden tai ei yhtään arvoa. Keskiasteen arvot toimitetaan Reduce-funktiolle iteraattorin avulla, mikä mahdollistaa sen, että voidaan käsitellä sellaisia määriä arvoja, jotka sellaisenaan eivät mahdu muistiin. [Dean and Ghemawat, 2008]

MapReduce-järjestelmä koostuu suuresta joukosta laskentaa suorittavia komponentteja. Järjestelmään syötetään suuri määrä tietoa. Syöte hajautetaan n:lle eri komponentille. Yksi komponenteista toimii Master-roolissa, ja sen tehtävänä on jakaa tehtävät muille komponenteille. Map-tehtäviä on m kappaletta ja Reduce-tehtäviä on r kappaletta. Master-komponentti valikoi järjestelmästä toimettomia komponentteja ja antaa niille joko Map-tehtävän tai Reduce-tehtävän. Map-tehtävän saanut komponentti lukee oman osuutensa syötteestä ja muodostaa siitä avain-arvo -pareja. Sitten se välittää jokaisen parin Map-funktiolle, joka tuottaa niistä keskiasteen avain-arvo -pareja, jotka puskuroidaan muistiin. Master-komponentti ilmoittaa Reduce-tehtävän saaneille komponenteille, mihin keskiasteen avain-arvo -parit on puskuroitu. Nämä komponentit puolestaan lukevat puskuroidun tiedon, lajittelevat sen avaimen mukaan siten, että kaikki saman avaimen omaavat parit on koottu yhteen. Sitten ne käyvät läpi lajitellun tiedon ja lähettävät Reduce-funktiolle jokaisen erilaisen avaimen ja listan sitä vastaavia arvoja. Reduce-funktio tuottaa r kappaletta tuloksia. [Dean and Ghemawat, 2008]

Master-komponentti pitää kirjaa siitä, mitä mikäkin komponentti tekee, siis mille komponenteille se on antanut Map-tehtävän ja mille Reduce-tehtävän. Lisäksi se pitää kirjaa tehtävien tiloista, siis onko tehtävä toimettomana, suoritettavana vai jo suoritettu. Master-komponentti kysyy säännöllisin väliajoin jokaiselta muulta toimijalta niiden tilaa. Jos se ei saa vastausta määrätyssä ajassa, se merkitsee kyseisen toimijan epäkuntoiseksi. Map- tai Reduce-tehtävä, jonka suoritus kyseisellä toimijalla on kesken, palautetaan alkutilaan ja voidaan antaa tehtäväksi jollekin toiselle komponentille. Loppuun suoritetut Map-tehtävät annetaan suoritettavaksi jollekin toiselle komponentille, koska niiden lopputulokseen ei päästä käsiksi. Sen sijaan loppuun suoritettuja Reduce-tehtäviä ei tarvitse suorittaa uudestaan. [Dean and Ghemawat, 2008]

Edellä kuvattujen ominaisuuksiensa vuoksi MapReduce tukee erinomaisesti itsekorjautuvuutta ja itseoptimoituvuutta. Koska se kykenee jakamaan epäonnistuneet tehtävät uudestaan muille kom-

ponenteille, komponenttien kaatuminen ei riko koko järjestelmää. Samoin mikäli syöte on suuri, järjestelmä kykenee jakamaan sen riittävän monelle rinnakkaisesti toimivalle komponentille käsiteltäväksi, ja suorituskyky pysyy sallituissa rajoissa.

### 4.7. Tile Style

Tile Style -arkkitehtuurissa hyödynnetään tietokoneverkoston laskentavoimaa tarkoituksena ratkaista NP-täydellisiä ongelmia ja sen taustalla on malli molekyylien välisistä sidoksista. Arkkitehtuurin etuja ovat tietoturva, vakaus ja skaalautuvuus. Arkkitehtuurin lähtökohtana on ns. laattajärjestelmä, jolla on todistetusti pystytty ratkaisemaan jokin tunnettu NP-täydellinen ongelma. Jokaista kyseisessä järjestelmässä olevaa laattaa kohti vastaava Tile Style -arkkitehtuuri ottaa käyttöön yksinkertaisen laattakomponentin yhdessä verkostoon kuuluvista tietokoneista, yhdistää kyseisen komponentin muihin verkoston komponentteihin halutun arkkitehtuurin mukaisesti ja sitten luo kaksi kopiota komponentista. Sama prosessi toistetaan molemmille kopioille niin kauan, että ongelma saadaan ratkaistua tai kunnes on riittävän todennäköistä, että ratkaisua ei ole olemassa. Laattojen kahdentaminen johtaa siihen, että haavoittuvuudet ja yksittäisten komponenttien kaatumiset eivät estä järjestelmän toimintaa. [Oreizy et al., 2008]

Tällaisessa laattajärjestelmässä yksittäiset komponentit voidaan ajatella neliönmuotoisina laattoina, joilla on neljä sivua, pohjoinen, etelä, itä ja länsi, sekä tilamuuttuja. Laatat voivat liittyä toisiinsa, jos niiden yhteen liittyvien sivujen nimikkeet ovat yhteensopivat. Esimerkiksi laatan länsisivuun voi liittyä vain sellainen laatta, jonka itäinen sivu sopii yhteen alkuperäisen laatan länsisivun kanssa. Arkkitehtuurin komponentit ovat laattajärjestelmän laattatyyppien instansseja. Tile Style -tyylillä tehdyssä järjestelmässä voi olla suuri määrä komponentteja, mutta erilaisten komponenttien määrä on kuitenkin suhteellisen pieni. Arkkitehtuuri muodostuu verkostosta, jonka solmut voivat koostua useista laatoista. Laatat voivat rekrytoida uusia laattoja liittymään verkostoon. Uuden laatan rajapintojen täytyy sopia yhteen verkostossa jo olevien komponenttien rajapintojen kanssa. [Brun and Medvidovic, 2007]

Komponentit voivat käyttäytyä kahdella eri tavalla. Ne voivat joko kommunikoida naapurilaattojen kanssa rekrytoidakseen sopivia uusia komponentteja järjestelmään tai raportoida ongelman ratkaisun käyttäjälle. Laattojen välinen vuorovaikutus koostuu komponentin sivuja koskevan tiedon vaihtamisesta. Komponentit voivat kertoa naapurikomponenteilleen oman tilansa tai sivujensa tiedon, mutta ei mitään muuta. [Brun and Medvidovic, 2007]

Tile Style -arkkitehtuurin etuja ovat diskreettiys, virheidensietokyky ja skaalautuvuus. Diskreettiydellä tarkoitetaan sitä, että hajautetussa järjestelmässä, johon Tile Style -arkkitehtuuri sopii mainiosti, verkoston solmut eivät pysty päättelemään syötettä, jonka se saa käsiteltäväkseen. Yksittäinen solmu tuntee osan syötteestä, mutta yksikään solmu ei pääse käsiksi koko syötteeseen. Virheidensietokyvyllä tarkoitetaan sitä, että vaikka joku verkoston solmuista menisi rikki tai toimisi virheellisesti, järjestelmän todennäköisyys tuottaa oikea ratkaisu ongelmaan on edelleen 100 prosenttia tai ainakin lähellä sitä ilman, että suoritusaika kasvaa merkittävästi. Tämä toteutuu Tile Sty-

le -arkkitehtuurissa, koska järjestelmässä voi olla useita samaa tyyppiä olevia komponentteja, jotka suorittavat samaa laskentaa. Jos jonkun toiminta häiriintyy, se ei vaikuta muihin vastaaviin komponentteihin, vaan ne tuottavat edelleen oikean ratkaisun. Järjestelmä on skaalautuva, mikäli verkostossa olevien solmujen vuorovaikutuksen määrä ei kasva syötteen kasvaessa. Laattajärjestelmän kokoonpanossa jokainen laatta vaatii tietyn määrän vuorovaikutusta liittyäkseen kokoonpanoon. Kun laatta on liittynyt järjestelmään, se voi osallistua ainoastaan kahden uuden laatan rekrytoimiseen, eli vuorovaikutuksen määrä on rajoitettu. [Brun and Medivdovic, 2007]

Koska Tile Style -arkkitehtuurin komponentit voivat rekrytoida uusia komponentteja liittymään järjestelmään, se tukee erinomaisesti itseoptimoituvuutta. Virheidensietokyvyn ja itseoptimoituvuutta tukevien ominaisuuksiensa ansiosta se soveltuu myös itsekorjautuvuuden toteuttamiseen. Arkkitehtuuri ei tosin tue komponenttien poistamista sellaisenaan, mutta sellaisia ominaisuuksia voidaan toteuttaa järjestelmään esimerkiksi Component Removal -suunnittelumallin avulla.

### 4.8. SASSY

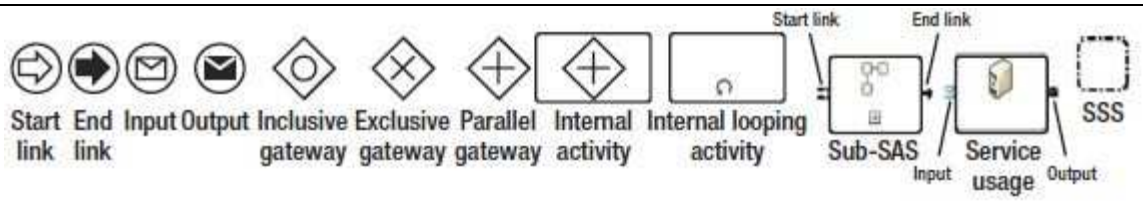
SASSY on kehys, joka tarjoaa puitteet itseohjautuvien järjestelmien rakentamiseen. Sen avulla voidaan toteuttaa itsekorjautuvia, itsejohtavia ja itseoptimoituvia järjestelmiä. SASSY (Self-architecting Software Systems) on malliperustainen lähestymistapa, jonka kohteena ovat dynaamiset asetukset, joissa järjestelmän vaatimukset voivat muuttua. Koko järjestelmän elinkaaren ajan SASSY ylläpitää optimaalista arkkitehtuuria, jonka avulla sekä toiminnalliset että ei-toiminnalliset vaatimukset toteutuvat. Arkkitehtuurin laatu ilmaistaan käyttäjän määrittelemän hyötyfunktion avulla ja se edustaa yhtä tai useampaa toivottua tavoitetta. SASSY:n tavoitteena on tarjota ympäristö, jossa palveluntarjoajat voivat tarjota toiminnallisesti samoja palveluja, mutta erilaatuisina ja eri hintaan. SASSY rajoittuukin vain palveluperustaisiin arkkitehtuureihin (SOA). [Menascé et al., 2011]

SASSY:ssa määritellään järjestelmän vaatimukset käyttäen visuaalista toimintojenmallinnuskieltä, josta SASSY sitten automaattisesti johtaa arkkitehtuurin, joka määrittää, mitkä palvelut järjestelmään kuuluvat. Haasteellisinta on se, miten määritellään alkuperäinen arkkitehtuuri, ja miten arkkitehtuuri mukautuu dynaamisesti ajonaikana ylläpitääkseen laatuvaatimukset. Tähän SASSY käyttää heuristiikkaa, jolla se generoi optimaalista arkkitehtuuria lähellä olevia arkkitehtuureja lieventääkseen ongelman monimutkaisuutta. [Menascé et al., 2011]

Alkuperäisen arkkitehtuurin generoiminen tapahtuu siten, että ensin kehitetään QoS-arkkitehtuurimalleja, kuten Replication, Load balancing ja Mediation [Menascé et al., 2010]. Nämä mallit yhdistetään parametrisoituihin analyyttisiin QoS-malleihin, jotka määrittävät, miten tietty malli vaikuttaa tiettyihin laatuominaisuuksien mittareihin. Lisäksi kehitetään mukautumismalleja, joiden mukaan järjestelmän arkkitehtuuri mukautuu nykyisestä arkkitehtuurista toiseksi ajonaikana. Lisäksi määritellään *palvelun toimintokaavioita* (service activity schema), joka ilmaisee järjestelmän vaatimukset. Nämä kaaviot liitetään järjestelmän laatuominaisuuksiin liittyviin tavoitteisiin, kuten turvallisuuteen tai suorituskykyyn. SASSY käyttää kaavioiden avulla määritettyjä vaatimuk-

sia generoidakseen peruspalveluarkkitehtuurin, joka koostuu rakenne- ja käyttäytymiskaavioista. Nämä kaaviot koostuvat komponenteista ja yhdistäjistä. Peruspalveluarkkitehtuurista SASSY johtaa optimaalisen arkkitehtuurin valitsemalla sopivimmat palveluntarjoajat ja soveltamalla QoS-arkkitehtuurimalleja. Tämä optimaalinen arkkitehtuuri määritetään analyyttisten QoS-mallien ja optimointitekniikoiden avulla, joiden tavoitteena on löytää lähellä optimaalista olevia vaihtoehtoja, jotka maksimoivat järjestelmästä saatavan hyödyn. SASSY yhdistää tarvittavat palvelut ja ottaa käyttöön ohjauslogiikan ja tuottaa näin optimaalisimman arkkitehtuurin ilmentymän. SASSY käsittelee muutokset siten, että se muokkaa palveluarkkitehtuuria. Muokkaus käynnistää tarkistetun arkkitehtuurin generoimisen ja ajonaikaisen mukautumisen uudeksi arkkitehtuuriksi. [Menascé et al., 2011]

Palvelun toimintokaavio on visuaalinen vaatimustenmäärittelykieli. Se on muodollisesti määritelty ja sillä voidaan generoida toimeenpantavia arkkitehtuurimalleja. Kuvassa 13 on sen elementit ja niiden määrittelyt.




Kuva 13. Palvelun toimintokaavion symbolit ja selitykset. [Menascé et al., 2011]

Kaavion muodostaminen tehdään siten, että ensin valitaan tarvittavat palvelunkutsut ja toiminnot. Kielessä on näille eri symbolit. Sitten niille määritellään järjestys, jossa ne suoritetaan. Tähän käytetään portteja, jotka ohjaavat tapahtumien virtaa. Tapahtumat edustavat viestejä, joita palvelukutsut vaihtavat keskenään. Lopuksi määritellään QoS-vaatimukset *palvelusekvenssiskenaarioiden* (service sequence scenario, SSS) avulla. Palvelusekvenssiskenaario on hyvinmuodostettu palvelun-toimintokaavion aligraafi. [Menascé et al., 2011]

*Järjestelmän palveluarkkitehtuuri* (system service architecture, SSA) tarjoaa rakenteellisia ja käyttäytymismalleja palveluperustaisille järjestelmille. Toisin kuin perinteisissä ohjelmistoarkkitehtuurimalleissa, joita käytetään yleensä suunnitteluvaiheessa, SASSY käyttää palveluarkkitehtuurimalleja ajonaikana. Palveluarkkitehtuurimalli on reaaliaikainen esitys järjestelmästä. Rakenteelliset mallit perustuvat laajennettavaan arkkitehtuurienkuvauskieleen, xADL:ään, jolla voidaan kuvata järjestelmän komponentit ja yhdistäjät. Käyttäytymismallit kuvaavat, sitä miten palveluinstanssit yhteistyössä täyttävät järjestelmän vaatimukset. Käyttäytymismallit perustuvat FSP-kieleen (Finite State Process). Toisin kuin monet muut automaatteja kuvaavat kielet, FSP tarjoaa paljon abstrakteja rakenteita, mikä mahdollistaa skaalautumisen kuvaamaan suurtenkin järjestelmien käyttäytymistä. [Menascé et al., 2011]

SASSY käyttää QoS-arkkitehtuurimalleja vähentääkseen arkkitehtuurien automaattiseen generoimiseen liittyviä ongelmia. QoS-arkkitehtuurimalleilla voidaan lisätä jotakin laatuominaisuutta

järjestelmään. Jokainen malli sisältää yhden tai useamman komponentin, jotka liitetään toisiinsa yhdistäjillä. Jokainen komponentti voidaan liittää yhteen tai useampaan palvelutyyppiin, joista yksi tai useampi palveluntarjoaja tuottaa instanssin. Lisäksi QoS-arkkitehtuurimalli sisältää yhden tai useampia QoS-mittareita ja vastaavan QoS-mallin. Esimerkiksi vikasietoisuutta lisäävä arkkitehtuurimalli vaikuttaa kahteen QoS-mittariin, saavutettavuuteen ja suoritusaikaan. [Menascé et al., 2011]

SASSY käyttää arkkitehtuurien määrittämiseen menetelmää, joka maksimoi järjestelmän hyötyfunktion arvon. Hyötyfunktio ilmaisee järjestelmän hyödyllisyyden sen attribuuttien arvojen funktiona. Esimerkiksi järjestelmän hyödyllisyys voi vähentyä nollaan, jos vasteajat kasvavat. Hyötyfunktioiden avulla voidaan myös määrittää laatuvaatimusten tasapainokohtia. Tasapainokohdalla tarkoitetaan sellaista tilannetta, jossa jokin arkkitehtuuriratkaisu vaikuttaa positiivisesti johonkin laatuominaisuuteen, mutta negatiivisesti johonkin toiseen. SASSY yhdistää kaikkien palvelusekvenssiskenaarioiden hyötyfunktiot yhdeksi globaaliksi hyötyfunktioksi, joka määrittää koko järjestelmän hyödyllisyyden. SASSY tarkkailee tämän hyötyfunktion arvoa ja käynnistää mukautumisen, mikäli sen arvo laskee alle ennalta määritellyn raja-arvon. [Menascé et al., 2011]

Mahdollisten uusien arkkitehtuurien määrä riippuu palveluntarjoajien, palveluinstanssien ja mahdollisten arkkitehtuurimallien määrästä ja voi olla hyvinkin suuri. Sen takia SASSY käyttää uuden optimaalisimman arkkitehtuurin määrittämiseen menetelmää, jossa lähdetään liikkeelle nykyisestä arkkitehtuurista ja generoidaan joukko niin sanottuja naapuriarkkitehtuureja korvaamalla komponentteja tai niiden yhdistelmiä QoS-arkkitehtuurimalleilla, jotka lisäävät haluttua laatuominaisuutta. Sitten jokaisen naapuriarkkitehtuurin jokaiselle palveluntarjoajalle suoritetaan lähes optimaalinen palveluiden jako, minkä jälkeen verrataan generoitujen arkkitehtuurien hyötyfunktioiden arvoja. Suurimman arvon omaavasta arkkitehtuurista aletaan taas generoida naapuriarkkitehtuureja, ja prosessia toistetaan, kunnes hyötyfunktion arvo ei enää kasva. [Menascé et al., 2011]

Mukautumismallit määrittelevät, miten järjestelmä siirtyy ajonaikaisesti konfiguraatiosta toiseen ilman, että sen toiminta häiriintyy. Mallit voidaan kuvata tilakaavioina, jotka määrittelevät komponenttien tilat ja siirtymät niiden välillä. Komponentti on aktiivinen, kun se osallistuu johonkin toiminnallisuuteen, ja siirtyy uinuvaan tilaan, kun se ei ole mukana missään toiminnassa eivätkä muut komponentit kommunikoi sen kanssa. Jokaiselle QoS-arkkitehtuurimallille on olemassa vastaava mukautumismalli, joka määrittelee, miten mallin komponentit sisällytetään konfiguraatioon. [Menascé et al., 2011]

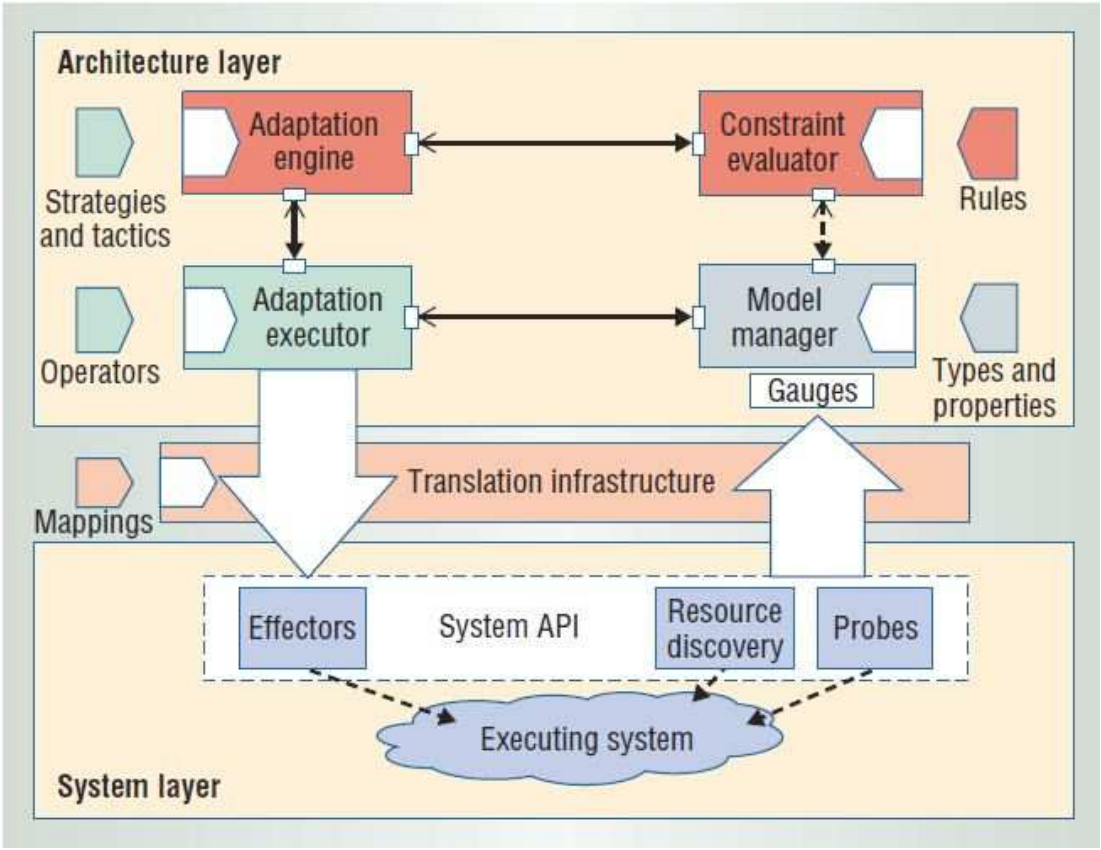
### 4.9. Rainbow

Rainbow on kehys, joka käyttää ohjelmistoarkkitehtuureja ja uudelleenkäytettävää rakennetta tukeakseen järjestelmien itseohjautuvuutta. Se on kehitetty tukemaan kaikenlaisia ohjelmistoarkkitehtuureja ja sen avulla voidaan toteuttaa itseohjautuvuutta kokonaan uuteen järjestelmään ja lisätä se jo olemassa olevaan järjestelmään. [Garlan et al., 2004]

Rainbow sisältää järjestelmän arkkitehtuurin mallin, jota tarkkaillaan ajonaikana. Malli kuvaa kaikki järjestelmän komponentit ja niiden väliset yhteydet. Lisäksi se voi kuvata järjestelmän kan-

nalta tärkeitä ominaisuuksia, kuten vasteaikaa tai suorituskykyä. Malli myös ylläpitää järjestelmän arkkitehtuurille asetettuja rajoitteita. Tällainen rajoite voi olla esimerkiksi se, että asiakas-palvelin -arkkitehtuurissa jokainen asiakas on yhteydessä vain yhteen palvelimeen. Järjestelmän täytyy jokaisen muutoksen jälkeen olla määriteltyjen rajoitteiden mukainen. [Garlan et al., 2004]

Rainbow:n komponentit ovat uudelleenkäytettäviä. Kehys voidaan jakaa kahteen osaan, mukautumisinfrastruktuuriin ja järjestelmäriippuvaiseen mukautumistietoisuuteen. Infrastruktuuri jaetaan järjestelmäkerrokseen, arkkitehtuurikerrokseen ja tulkkikerrokseen. Infrastruktuuri sisältää ne komponentit, jotka ovat uudelleenkäytettäviä. Kuvassa 14 on Rainbow:n infrastruktuuri.




Kuva 14. Rainbow:n infrastruktuuri. [Garlan et al., 2004]

Järjestelmäkerroksessa määritellään rajapinta, jonka avulla järjestelmään päästään käsiksi. Kerrokseen kuuluu kolme erilaista mekanismia, joilla on omat tehtävänsä. Mittausmekanismi tarkkailee ja mittaa järjestelmän tilaa, ja näin kerätty tieto voidaan joko lähettää anturien avulla eteenpäin tai tietoa tarvitseva komponentti voi kysyä sitä antureilta. Resurssienlöytämismekanismilta voidaan pyytää uusia resursseja esimerkiksi resurssin tyypin tai muun kriteerin perusteella. Toimeenpanomekanismin tehtävänä on suorittaa järjestelmän mukauttaminen. [Garlan et al., 2004]

Arkkitehtuurikerroksessa mittarit keräävät tietoa antureilta ja tiedon perusteella päivittävät järjestelmän arkkitehtuurimallin attribuutteja. Mallin ohjain käsittelee arkkitehtuurimallia ja sen kautta pääsee käsiksi malliin. Rajoitteiden arvioijan tehtävänä on tarkistaa säännöllisin väliajoin mallin tila

ja käynnistää mukautuminen, mikäli mallin tila ei vastaa vaatimuksia. Mukautumismoottori päättää, miten järjestelmän täytyy mukautua ja suorittaa tarvittavan mukautumisen. [Garlan et al., 2004]

Tulkkikerroksen tehtävänä on huolehtia tiedon välittämisestä järjestelmästä mallille ja mallilta takaisin järjestelmään. Se sisältää kokoelman erilaisia muunnoksia, joilla erilaiset tietoformaatit voidaan kääntää toisen kerroksen ymmärtämään muotoon, esimerkiksi arkkitehtuuritason elementin tunniste IP-osoitteeksi tai arkkitehtuuritason muutosoperaattori järjestelmätason operaatioksi. [Garlan et al., 2004]

Jotta järjestelmään voidaan lisätä itseohjautuvuutta käyttäen niitä toimintoja, joita mukautumisinfrastruktuuri tarjoaa, täytyy infrastruktuurin räätälöimiseksi käyttää hyväksi tietoa järjestelmäkohtaisesta mukautumisesta. Tieto sisältää kohdejärjestelmän toiminnallisen mallin, joka määrittelee parametrit, kuten komponenttien tyypit, sekä attribuutit, käyttäytymistä koskevat rajoitteet ja mukautumisstrategiat. [Garlan et al., 2004]

Samoin kuin Rainbow:n uudelleenkäytettävä rakenne auttaa vähentämään kustannuksia lisättäessä itseohjautuvuutta järjestelmiin se mahdollistaa myös järjestelmien arkkitehtuurien yhtäläisyyksien löytämisen, jolloin voidaan kapseloida mukautumistietämystä useille järjestelmäluokille. Tämän Rainbow tekee mukauttamalla arkkitehtuurityylin käsitettä. Perinteisesti arkkitehtuurityylejä käytetään ilmaisemaan järjestelmäriippuvaista tietoa. Arkkitehtuurityyleissä on tyypillisesti neljän tyyppisiä entiteettejä: komponentti- ja yhdistäjätyyppejä, rajoitteita, ominaisuuksia ja analyyseja. Komponenttityyppejä ovat esimerkiksi tietokanta, asiakas, palvelin ja suodatin. Yhdistäjätyyppejä ovat esimerkiksi väylä, SQL, HTTP ja RPC. Rajoitteet määrittelevät, millaiset komponenteista ja yhdistäjistä muodostetut rakenteet ovat sallittuja. Rajoitteet voivat esimerkiksi määrätä, että tietovuoarkkitehtuurissa väylät ja suodattimet eivät saa muodostaa syklejä. Ominaisuuksilla tarkoitetaan komponentti- ja yhdistäjätyyppien attribuutteja ja ne sisältävät analyyttista, käyttäytymis- tai semanttista informaatiota. Analyyseja voidaan suorittaa järjestelmille, jotka on rakennettu tietyn arkkitehtuurityylin mukaisesti. Esimerkiksi asiakas-palvelin-arkkitehtuurin mukaiselle järjestelmälle voidaan suorittaa suorituskykyanalyysi. [Garlan et al., 2004]

Nämä entiteetit sisältävät järjestelmän staattiset ominaisuudet. Rainbow laajentaa arkkitehtuurityylin käsitettä lisäämällä siihen kaksi uutta entiteettiä: mukautumisoperaattorit ja mukautumisstrategiat. Mukautumisoperaattorit määrittelevät joukon arkkitehtuurityylistä riippuvaisia toimintoja, joita järjestelmän ohjausrakenne voi suorittaa muuttaakseen järjestelmän konfiguraatiota. Esimerkiksi asiakas-palvelin -arkkitehtuurissa voisi olla mukautumisoperaattori RemoveClient, joka poistaisi jonkin asiakaskomponentin. Mukautumisstrategiat määrittelevät, millaisia muutoksia järjestelmään voidaan soveltaa, jotta se saadaan epätoivotusta tilasta toivottuun tilaan. Asiakas-palvelin -arkkitehtuurissa voisi olla määritelty esimerkiksi rajoite, joka määrää palvelimen suurimman vasteajan. Jos vasteaika ylittyy, mukautumisstrategia voisi jakaa palvelimelle tulevia palvelupyyntöjä muille vastaaville palvelimille. Mukautumisstrategiat määritellään käyttäen operaattoreita ja ominaisuuksia. [Garlan et al., 2004]

Mukautumisstrategiat liittyvät aina tiettyyn järjestelmän ominaisuuteen. Ominaisuuteen liittyy joukko järjestelmän ei-toiminnallisia vaatimuksia, ja se määrittelee joukon järjestelmän attribuutteja, joihin mukautuminen kohdistuu ja siten myös joukon mukautumisstrategioita. Järjestelmän ominaisuudet muodostavat järjestelmän arkkitehtuurityylin attribuuttien osajoukon. Rainbow:n järjestelmäriippuvaisen mukautumistietoisuuden uudelleenkäytettävyys riippuu siitä, kuinka samanlaisia järjestelmät ovat arkkitehtuurityyleiltään ja ominaisuuksiensa osalta. [Garlan et al., 2004]

# 5. Suunnittelumallit

Suunnittelumallit ovat abstrakteja uudelleenkäytettäviä ratkaisuja usein esiintyviin ongelmiin. Gamman ja muiden [1995] mukaan suunnittelumallit koostuvat neljästä elementistä: nimestä, ongelmasta, ratkaisusta ja seurauksista. Jokaisella mallilla on nimi, jota käytetään kuvaamaan kolmea muuta elementtiä. Ongelma kuvaa tilanteen, jossa suunnittelumallia käytetään. Ratkaisu kuvaa ne elementit, joita tarvitaan ongelman korjaamiseksi, ja seuraukset kuvaavat sen tilan, joka syntyy kun mallia on sovellettu ongelman ratkaisemiseksi.

Tässä luvussa esittelen suunnittelumalleja, joita voidaan käyttää itseohjautuvien järjestelmien suunnittelussa. Kohdassa 5.1. esittelen malleja, jotka tukevat itseohjautuvuutta yleisesti. Mukana on malleja, joita voidaan käyttää järjestelmän tai sen ympäristön tarkkailemiseen, malleja, joilla voidaan tehdä päätös, miten järjestelmän tulisi muuttaa itseään, sekä malleja, joita voidaan käyttää järjestelmän komponenttien välisten yhteyksien muuttamiseen. Kohdassa 5.2. esittelen kontekstitietoisuutta tukevia malleja ja kohdassa 5.3. itseorganisoituvuutta tukevia malleja. Kohdassa 5.4. käsittelen uudelleenkonfiguroitumismalleja.

## 5.1. Itseohjautuvuutta tukevia suunnittelumalleja

Ramirez ja Cheng [2010] esittelevät erilaisia suunnittelumalleja itseohjautuvien järjestelmien suunnittelun avuksi. He ovat tutkineet kymmeniä itseohjautuvia järjestelmiä käsitteleviä artikkeleita ja projekteja, joista he ovat tunnistaneet erilaisia itseohjautuvuutta tukevia suunnittelumalleja. Näitä malleja he ovat soveltaneet itseohjautuvan uutispalvelun suunnittelussa.

Ramirez ja Cheng [2010] jakavat löytämänsä suunnittelumallit kolmeen kategoriaan, tarkkailumalleihin, päätöksentekomalleihin ja uudelleenkonfiguroitumista tukeviin malleihin. Tarkkailumalleja on kolme: Sensor Factory, Reflective Monitoring ja Content-based Routing. Päätöksentekomalleja ovat Adaptation Detector, Case-based Reasoning, Divide and Conquer, Arhitecture-based ja Tradeoff-based. Uudelleenkonfiguroitumista tukevia malleja ovat Component Insertion, Component Removal, Server Reconfiguration ja Decentralized Reconfiguration.

Sensor Factory -mallin tarkoitus on ottaa käyttöön sensoreita, jotka keräävät tietoa järjestelmän komponenteilta. Valvottavien komponenttien täytyy toteuttaa AbstractSensor-rajapinta, jonka kautta niiltä voidaan kysyä tarvittavaa tietoa. SensorFactory-luokka hallitsee sensorikokoelmaa siten, että sensorit eivät ole suoraan yhteydessä komponentteihin ja asiakkaisiin. Lisäksi Registry-luokka määrittelee, onko sensorien keräämä data jaettavaa vai ei. Malli mahdollistaa uusien sensorien liittämisen järjestelmään ja saattaa vähentää resurssien käyttöä jakamalla sensorien keräämää informaatiota, mutta saattaa samalla heikentää suorituskykyä sensorien ja asiakkaan väliin tulevan ylimääräisen kerroksen vuoksi. [Ramirez and Cheng, 2010]

Reflective Monitoring -mallissa komponenttiin liitetään mekanismi, jonka avulla se voi sekä tarkkailla omaa sisäistä tilaansa että tarjota sensoreille rajapinnan, jonka kautta ne voivat pyytää siltä informaatiota. Malli hyödyntää reflektiota kiertääkseen komponentin kapseloinnin ja tarjotakseen metodeja, joiden avulla voidaan päästä käsiksi komponentin attribuuttien arvoihin. Mallin

avulla voidaan erottaa toisistaan tarkkailun mahdollistava rajapinta ja komponentin toiminnallinen logiikka. [Ramirez and Cheng, 2010]

Content-based Routing -malli tarjoaa ratkaisun tilanteeseen, jossa usea asiakaskomponentti tarvitsee samaa informaatiota, mikä vaikuttaa kyseistä informaatiota tarjoavan komponentin toimintaan. Sen sijaan, että asiakaskomponentit pyytäisivät tietoa suoraan sensoreilta, käytetään monta-moneen -tyyppistä tilaaja-tuottaja -mallia, joka ottaa tiedon sensorilta ja jakaa sen asiakaskomponenteille. Asiakaskomponentit määrittelevät, minkälaisista tapahtumista ne ovat kiinnostuneita, ja kun tilaaja-tuottaja -malli havaitsee sensoreiden tuottavan kyseisenlaista dataa, se toimittaa tiedon asiakaskomponenteille. [Ramirez and Cheng, 2010]

Adaptation Detector -mallissa sensoreilta saatavat tietovirrat liitetään arvoihin, jotka havainnollistavat järjestelmän odotetun ja havaitun käyttäytymisen eroavaisuutta. Jos poikkeama on tarpeeksi suuri, käynnistetään päätöksentekoprosessi, jossa ensin valitaan sopiva uudelleenkonfiguroitumistapa. Mallin avulla voidaan erottaa tarkkailuprosessi ja päätöksentekoprosessi toisistaan. [Ramirez and Cheng, 2010]

Case-based Reasoning -mallissa yhdistetään ennalta määritellyt tarkkailun tulokset tiettyihin uudelleenkonfiguroitumisohjeisiin. Malli soveltuu käytettäväksi tilanteissa, joissa erilaiset uudelleenkonfiguroitumista vaativat tilanteet voidaan helposti ja luotettavasti erottaa toisistaan ja joissa päätöksentekologiikka on yksinkertainen ja ilmaistavissa if-else -rakenteilla. Mallin eduksi voidaan laskea se, että sen avulla erotetaan toimintalogiikka päätöksentekologiikasta. [Ramirez and Cheng, 2010]

Divide and Conquer -mallissa monimutkainen uudelleenkonfiguroitumismekanismi hajotetaan useammaksi yksinkertaiseksi mekanismiksi. Malli määrittelee ensin eri mekanismien riippuvuussuhteet, luo sitten järjestyksen, joka säilyttää riippuvuudet, ja lopuksi suorittaa rinnakkain nämä mekanismit. [Ramirez and Cheng, 2010]

Architecture-based -malli perustuu arkkitehtuurimalleihin, joilla kuvataan sekä järjestelmän nykyinen arkkitehtuuri että mahdolliset muut arkkitehtuurit. Suunnittelumallissa tutkitaan jatkuvasti järjestelmän nykyistä tilaa siltä varalta, että järjestelmän ominaisuudet ovat muuttuneet. Mikäli näin on käynyt, käydään läpi muut mahdolliset arkkitehtuurimallit ja valitaan niistä sopivin, joka sitten otetaan käyttöön. [Ramirez and Cheng, 2010]

Tradeoff-based -mallissa arvioidaan mahdollisia uusia konfiguraatioita, jotka pisteytetään sen mukaan, miten järjestelmän tavoitteet toteutuvat niissä. Vaihtoehdoista valitaan sitten parhaan tuloksen saanut konfiguraatio. Suunnittelumallin etu on, että vaihtoehdoista valitaan se, jossa järjestelmän laatuominaisuudet ovat tasapainossa. Huonona puolena pidetään sitä, että jokaiseen laatuominaisuuteen täytyy liittää hyötyfunktio. Hyötyfunktiot ovat usein sovellusriippuvaisia, joten uudelleenkäyttö on vaikeaa. [Ramirez and Cheng, 2010]

Component Insertion -mallissa alustetaan uusi komponentti joko oletustilaan tai johonkin aikaisemmin tallennettuun tilaan. Ennen uuden komponentin lisäystä ne komponentit, jotka ovat vuorovaikutuksessa uuden komponentin kanssa, asetetaan passiiviseen tilaan. Kun uusi komponentti on

lisätty järjestelmään ja yhdistetty muihin komponentteihin, passiiviset komponentit muutetaan taas aktiivisiksi. Suunnittelumallin avulla voidaan estää epäjohdonmukaisten transaktioiden suorittaminen, mutta useiden komponenttien yhtäaikainen passivoiminen voi väliaikaisesti heikentää järjestelmän suorituskykyä. [Ramirez and Cheng, 2010]

Component Removal -malli toimii samaan tapaan kuin komponentin lisäyskin. Ennen poistoa kaikki transaktiot, joihin poistettava komponentti osallistuu, suoritetaan loppuun, jotta järjestelmän toiminta ei häiriinny, ja sekä poistettava komponentti että ne komponentit, jotka ovat yhteydessä siihen, asetetaan passiiviseen tilaan, jolloin poistettava komponentti eristetään muusta järjestelmästä eikä se voi osallistua transaktioihin. Samoin kuin Component Insertion -mallissa, myös Component Removal -mallissa komponenttien passivoiminen voi heikentää järjestelmän prosessointikykyä. [Ramirez and Cheng, 2010]

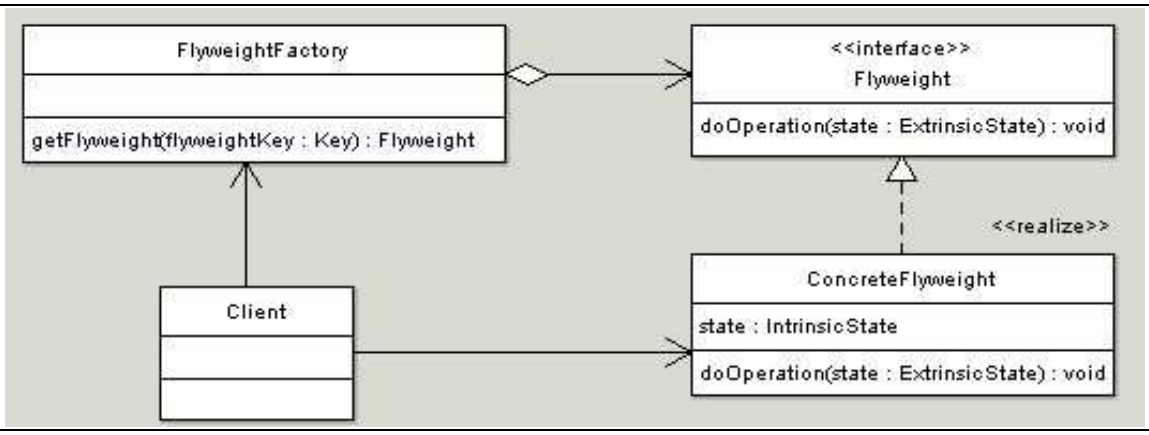
Server Reconfiguration -mallin tarkoituksena on konfiguroida palvelin uudestaan ajonaikana ilman, että asiakkaiden lähettämiä kutsuja hukataan. Malli hallitsee niiden komponenttien operatiivista tilaa, jotka muodostavat asiakas-palvelin -arkkitehtuurin. Se puskuroi asiakkailta tulevat kutsut, ja ne suoritetaan, kun uudelleenkonfiguraatio on valmis. Mallin avulla palvelin on oikeassa tilassa sekä ennen konfiguraatiota, sen aikana että sen jälkeen. [Ramirez and Cheng, 2010]

Decentralized Reconfiguration -mallia voidaan käyttää silloin, kun järjestelmässä ei ole keskitettyä ohjaavaa komponenttia. Tällöin jokainen komponentti osallistuu uudelleenkonfiguroitumiseen ja vastaa itsensä alustamisesta, asentamisesta sekä itsensä ja muiden komponenttien välisten yhteyksien katkaisemisesta. Mallin avulla saadaan muodostettua yhtenäinen konfiguroitumisprotokolla ja säilytetään järjestelmän johdonmukaisuus, mutta samalla uuden konfiguraation oikeellisuuden varmistaminen vaikeutuu, sillä jokainen komponentti on vastuussa itsestään. [Ramirez and Cheng, 2010]

### 5.2. Kontekstitietoisuutta tukevia suunnittelumalleja

Riva ja muut [2006] esittelevät malleja, jotka tukevat järjestelmien kontekstitietoisuutta. He ovat tutkineet erilaisia ympäristöään tarkkailevia järjestelmiä ja etsineet niissä käytettyjä suunnittelumalleja. Tunnistetut mallit on jaettu kahteen ryhmään: niihin, jotka on esitelty Gamman ja muiden teoksessa [1995], sekä uusiin malleihin. Ennestään tunnettuja malleja ovat Flyweight, Hybrid Mediator-Observer ja Strategy. Uusia malleja ovat Flexible Context Processing ja Enactor.

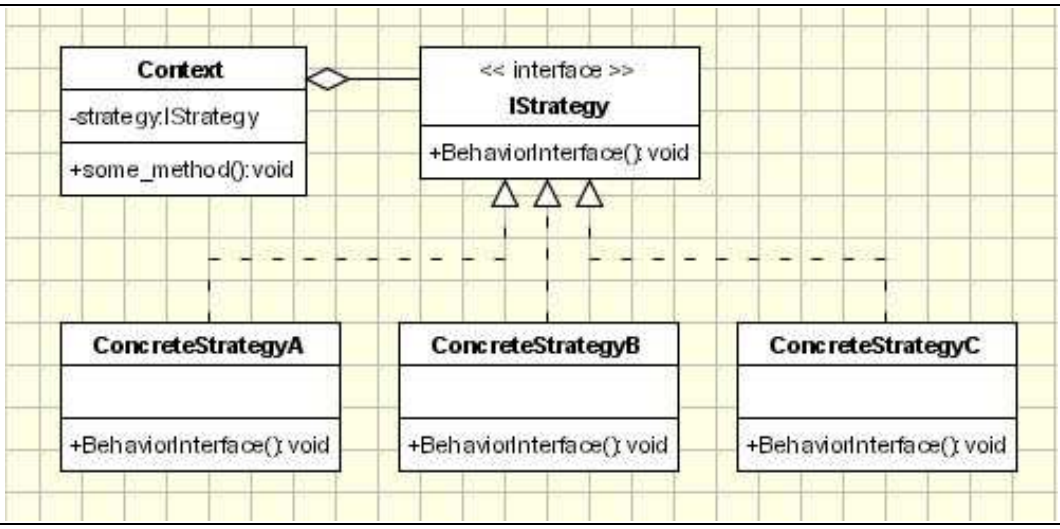
Flyweight-mallia käytetään sekä havainnointiin että komponenttien säätelyyn. Se on hyödyllinen silloin, kun täytyy hallita suurta määrää sensoreita ja toimeenpanijoita. Flyweight on eri komponenttien yhteinen jäsen, jota voidaan käyttää samanaikaisesti useassa eri tilanteessa. Sen hyöty perustuu sisäisen ja ulkoisen tilan erottamiseen. Sisäinen tila koostuu tiedoista, jotka ovat riippumattomia sovelluslogiikasta, mikä mahdollistaa sisäisen tilan jakamisen. Ulkoinen tila on riippuvainen sovelluslogiikasta ja vaihtelee sen mukaan, minkä vuoksi ulkoista tilaa ei voi jakaa komponenttien kesken. Asiakasolioiden tehtävä on tarvittaessa antaa ulkoinen tila Flyweight-oliolle. Kuvassa 15 on Flyweight-mallin luokkakaavio. [Riva et al., 2006]




Kuva 15. Flyweight-mallin luokka kaavio. [OODesign, 2012]

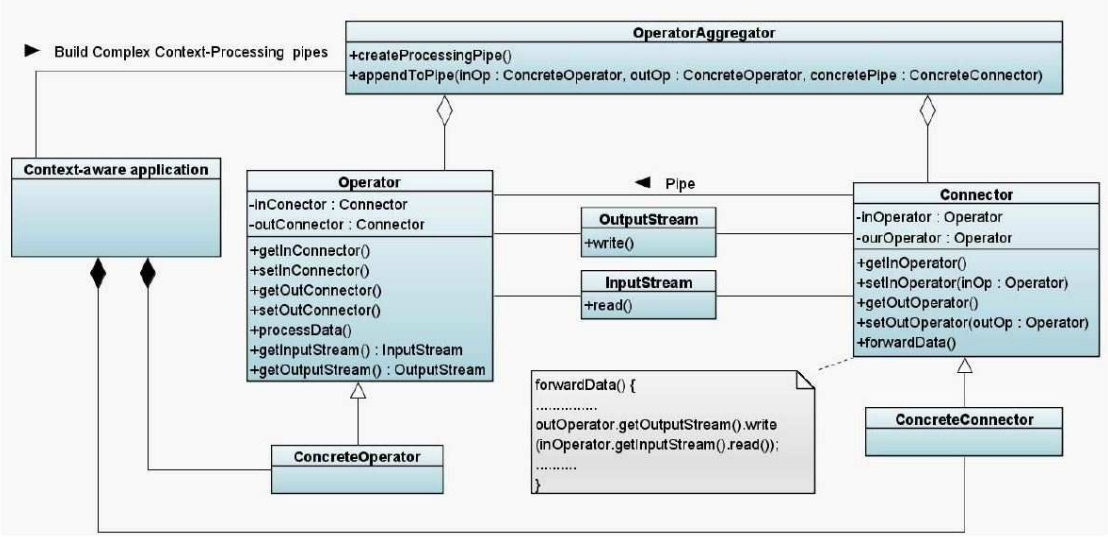
Hybrid Mediator-Observer -mallissa yhdistyy kaksi Gamman ja muiden [1995] esittelemää mallia. Observer-mallia käytetään järjestelmän kontekstissa tapahtuvien muutosten havaitsemiseen. Kontekstiparametrien ja Observer-komponenttien välillä on sellainen yksi moneen -suhde, että kun parametri muuttuu, siitä ilmoitetaan Observer-komponenteille. Parametrien ja tarkkailijoiden välisistä riippuvuuksista voi tulla erittäin monimutkaisia, kun parametrien tyyppi ja määrä muuttuu tai kun tarkkailijoiden määrä kasvaa. Riippuvuuksien vähentämiseksi käytetään Mediator-mallia, jonka avulla erotetaan toisistaan tarkkailijat ja parametrit. Yhdessä Observer- ja Mediator-mallit muodostavat Hybrid Mediator-Observer -mallin. [Riva et al., 2006]

Strategy-mallia voidaan käyttää, kun järjestelmän täytyy käyttäytyä eri tavoin eri tekijöiden mukaan. Tällöin järjestelmä sisältää säännöstön, joka määrää, miten kukin tekijä vaikuttaa järjestelmän käyttäytymiseen. Järjestelmä voi esimerkiksi räätälöidä käyttäytymistään käyttäjien, ympäristön tai laitteiden mukaan. Strategy-mallin avulla mahdollistetaan kontekstista riippuvan käyttäytymisen muunneltavuus. Järjestelmään voidaan lisätä uusia käyttäytymisstrategioita tai vanhoja voidaan muuttaa. Kuvassa 16 on esitetty Strategy-mallin luokkakaavio. [Riva et al., 2006]



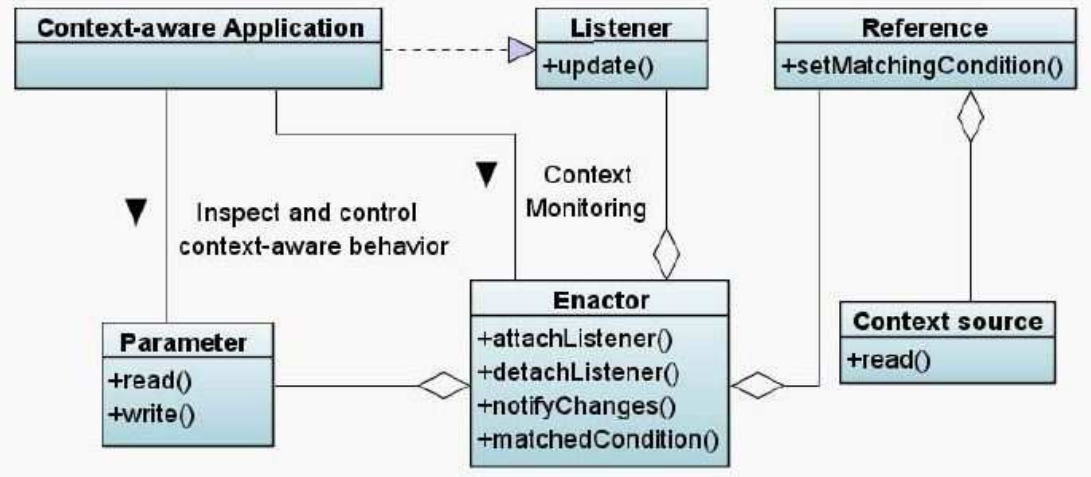

Kuva 16. Strategy-mallin luokkakaavio. [OODesign, 2012]

Flexible Context Processing -mallin tarkoituksena on erottaa kontekstin prosessointi varsinaisesta sovelluslogiikasta ja tarjota siten elementtejä, joita voidaan käyttää uudestaan. Mallissa Operator-oliot lukevat tietoa syötevirrasta, prosessoivat sen ja tulostavat sen tulostevirtaan. Connector-oliot muuntavat yhden Operator-olion tulostevirran toisen Operator-olion syötevirraksi. Sovellus antaa Operator- ja Connector-komponentit OperatorAggregator-komponentille, joka järjestää ne putkeksi, jossa tieto prosessoidaan. Mallin luokkakaavio on kuvassa 17. [Riva et al., 2006]




Kuva 17. Flexible Content Processing -mallin luokkakaavio. [Riva et al., 2006]

Enactor-mallissa ohjaus, tarkkailu ja toimeenpano erotetaan toisistaan. Enactor-komponentilla on kolme jäsentä: Reference-, Parameter- ja Listener-komponentit. Enactor-komponentti saa tietoa kontekstista Reference-komponentin kautta. Se prosessoi tiedon ja poimii siitä olennaisen tiedon parametreiksi. Listener-komponentti saa tiedon kaikista Enactor-komponentin tapahtumista, kuten Reference-komponentilta saadusta syötteestä tai toiminnoista, jotka Enactor käynnistää. Parameter-komponentin tehtävänä on ohjaus. Kuvassa 18 on Enactor-mallin luokkakaavio.




Kuva 18. Enactor-mallin luokkakaavio. [Riva et al., 2006]

### 5.3. Itseorganisoitumista tukevia malleja

Babaoglu ja muut [2006] esittelevät suunnittelumalleja, jotka on johdettu biologiasta ja joita voidaan soveltaa hajautettujen järjestelmien suunnittelussa. Tällaisia malleja ovat Plain Diffusion, Stigmergy, Replication, Chemotaxis ja Reaction-Diffusion. Jokaisessa näistä malleista hajautettu järjestelmä esitetään solmujen muodostamana verkostona. Jokaisella solmulla on joukko naapureita, joille se voi lähettää viestejä. Naapureiden joukkoa kutsutaan solmun näkymäksi. Solmuverkosto voidaan kuvata suunnattuna graafina siten, että jos solmu j on solmun i näkymässä, graafissa on suunnattu kaari solmusta i solmuun j.

Plain Diffusion -malli perustuu viestinvälitykseen ja sitä voidaan soveltaa kahdenlaisten ongelmien ratkaisemiseen. Ensimmäisessä tapauksessa oletetaan, että jokainen graafin solmuista sisältää jonkin numeerisen arvon. Tarkoituksena on saattaa järjestelmä sellaiseen tilaan, jossa jokainen solmuista sisältää arvojen keskiarvon. Myös toisessa ongelmatapauksessa solmut sisältävät jonkin arvon. Tavoitteena on lisätä jokaiseen solmun linkkiin kerroin, joka on verrannollinen arvon muutokseen, kun linkkiä seurataan, toisin sanoen positiivinen arvon kasvaessa ja negatiivinen arvon pienentyessä. Ensimmäinen ongelma ratkaistaan siten, että säännöllisin väliajoin jokainen graafin solmuista vähentää jokaiselle linkille määrätyn osuuden sisältämästään arvosta ja lähettää sen eteenpäin kyseistä linkkiä pitkin. Kun solmu vastaanottaa arvon, se lisää sen nykyiseen arvoonsa. Tällöin solmujen sisältämien arvojen summa pysyy vakiona. Samalla generoidaan linkkien kertoimet siten, että jos tiettyä linkkiä pitkin tulee positiivinen virta, linkki johtaa korkean arvon alueelle. [Babaoglu et al., 2006]

Replication-malli sopii kolmenlaisten ongelmien ratkaisemiseen. Ensimmäisessä tapauksessa tietty solmu saa uutta tietoa. Tavoitteena on, että sama tieto levitetään muillekin solmuille. Toisessa tapauksessa jokaiselle solmulle annetaan jokin numeerinen arvo. Tavoitteena on saada kaikkiin solmuihin arvojen maksimi. Kolmannessa tapauksessa solmut sisältävät jotakin informaatiota. Tarkoituksena on löytää solmu, jonka sisältämä informaatio vastaa tiettyä kyselyä. Replication-mallissa solmut vastaanottavat viestejä naapureiltaan ja lähettävät osan viesteistä eteenpäin sovelluksesta riippuvaisten sääntöjen mukaan. Ensimmäinen ongelmatapauksista ratkeaa mallin avulla siten, että solmut kopioivat kaiken uuden tiedon naapureilleen. Toinen tapaus ratkeaa siten, että solmut tallentavat paikallisesti saamansa maksimiarvon ja lähettävät sen eteenpäin naapureilleen. Kolmas tapaus voidaan ratkaista siten, että mallia käytetään hakukyselyihin. Solmut lähettävät kyselyn eteenpäin naapurisolmuilleen niin kauan kunnes kyselyä vastaava informaatio löytyy. [Babaoglu et al., 2006]

Stigmergy-malli soveltuu kahdenlaisten ongelmatapausten ratkaisemiseen. Ensimmäisessä ongelmatapauksessa oletetaan, että solmujen välisillä yhteyksillä on arvot painotetun graafin kaarien tapaan. Valitaan kaksi solmua ja tavoitteena on löytää lyhin reitti näiden solmujen välillä. Toisessa ongelmatapauksessa jokainen solmu sisältää joukon erilaisia objekteja, joilla on tietty ominaisuus. Tarkoituksena on jakaa objektit solmuille siten, että ne objektit, joilla on sama ominaisuus, ovat samassa solmussa. Stigmergy-mallissa jokainen solmu sisältää joukon stigmergisiä muuttujia. Sol-

mut generoivat viestejä ja lähettävät niitä naapureilleen sovelluksessa määriteltyjen sääntöjen mukaan. Viestin vastaanotto käynnistää solmussa toiminnon, joka riippuu saadun viestin sisällöstä ja solmun sisältämistä stigmergisistä muuttujista. Toiminto voi olla esimerkiksi stigmergisten muuttujien tai viestin sisältämän informaation päivittäminen tai viestin välittäminen eteenpäin. Koska muutokset stigmergisissä muuttujissa ovat pysyviä, se vaikuttaa siihen miten seuraavat viestit käsitellään. Lyhimmän reitin ongelma ratkeaa siten, että reitin alkusolmu lähettää toistuvia viestejä tavoitteenaan löytää loppusolmu. Viestin reittiin vaikuttavat välissä olevien solmujen stigmergiset muuttujat, jotka päivitetään pitämään kirjaa siitä, miten pitkä reitti loppusolmuun arvioidaan olevan. Toinen ongelma sen sijaan ratkeaa siten, että solmun sisältämät objektit ja niiden ominaisuudet ovat sen stigmergisiä muuttujia, ja myös viestit sisältävät objekteja. Nämä objektit vaikuttavat siihen, jääkö saapuvan viestin sisältämä tietty toinen objekti tiettyyn solmuun vai lähetetäänkö se eteenpäin naapurisolmulle. [Babaoglu et al., 2006]

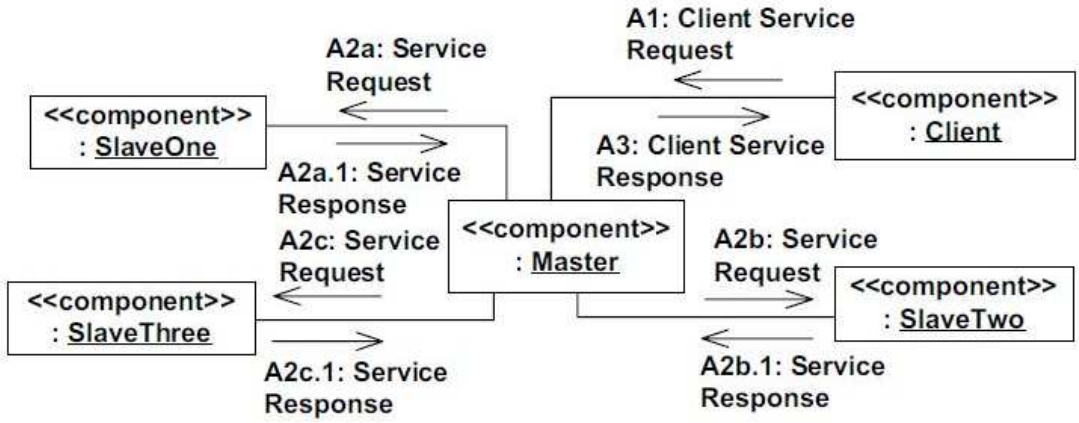
Chemotaxis-mallia voidaan käyttää tapauksessa, jossa täytyy löytää lyhyt reitti tietystä solmusta sellaisille alueille, joissa leviävän aineen pitoisuus on suurin. Ongelma ratkeaa siten, että alkusolmusta lähtevistä kaarista valitaan se, jolla on suurin kerroin. Näin jatketaan kunnes tavoite on saavutettu. [Babaoglu et al., 2006]

### 5.4. Uudelleenkonfiguroitumismalleja

Gomaa ja Hussein [2004] käsittelevät uudelleenkonfiguroitumismalleja, joilla tarkoitetaan sellaisia malleja, jotka määrittelevät, miten arkkitehtuurin tai suunnittelumallin muodostavat komponentit toimivat yhdessä muuttaakseen ohjelmiston konfiguraatiota. Gomaan ja Husseinin [2004] mukaan jokaiselle suunnittelumallille voidaan muodostaa uudelleenkonfiguroitumismalli. Tässä esitellään uudelleenkonfiguroitumismallit Master-Slave, Centralized Control, Client/Server ja Decentralized Control -malleille.

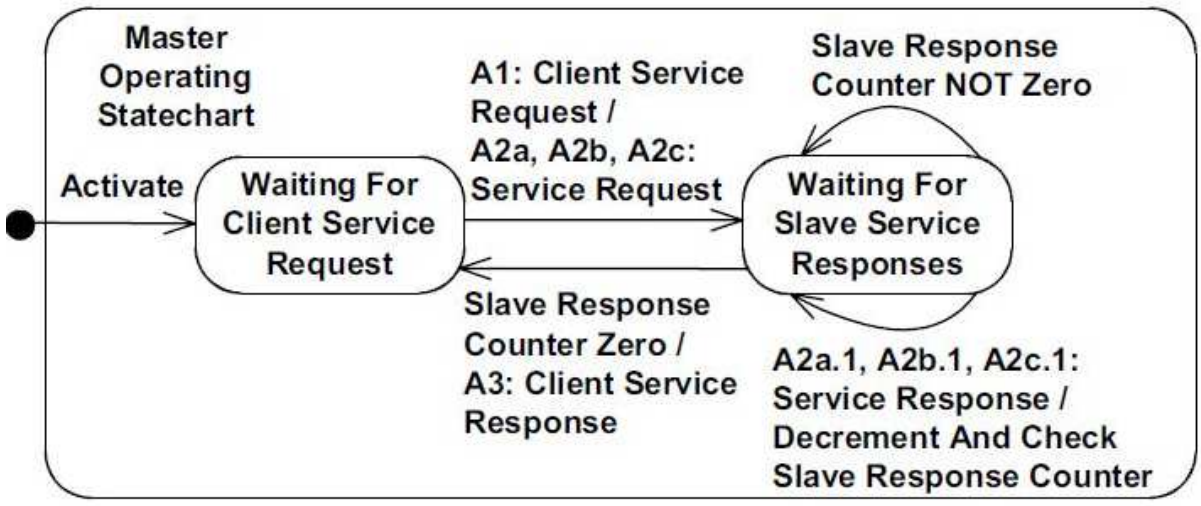
Master-Slave -suunnittelumallissa Master-komponentti vastaanottaa Client-komponentilta palvelupyyntöjä, jotka se jakaa Slave-komponenteille. Slave-komponentit käsittelevät pyynnöt ja lähettävät vastauksen Master-komponentille, joka sitten toimittaa vastauksen Client-komponentille. Kuva 19 kuvaa komponenttien vuorovaikutusta. Uudelleenkonfiguroitumismallissa sekä Master-komponentti että Slave-komponentit voidaan poistaa tai vaihtaa toiseen. Tässä mallissa komponentti voi olla viidessä erilaisessa tilassa: passiivinen, aktiivinen, toimeton, passivoituva tai kuittausta odottava. Komponentti on passiivisessa tilassa silloin, kun se ei osallistu mihinkään sen itsensä aloittamaan toimintoon eikä aloita uusia toimintoja. Se siirtyy toimettomaan tilaan, jos se on passiivinen, jos se ei osallistu mihinkään toimintoon ja muut komponentit eivät ole aloittaneet eivätkä aloita sellaisia toimintoja, joihin se osallistuu. Komponentti on passivoituvassa tilassa, kun se irtautuu sellaisista toiminnoista, jotka se on aloittanut tai joissa se on muuten ollut osallisena. Kuittausta odottavassa tilassa komponentti on silloin, kun se on lähettänyt sen kanssa vuorovaikutuksessa oleville komponenteille viestin, jossa se kertoo muuttuvansa passiiviseksi, ja odottaa myöntävää vas-

tausta. Komponentti on aktiivinen, jos se osallistuu johonkin toimintoon. [Gomaa and Hussein, 2004]




Kuva 19. Master-Slave -suunnittelumallin vuorovaikutuskaavio. [Gomaa and Hussein, 2004]

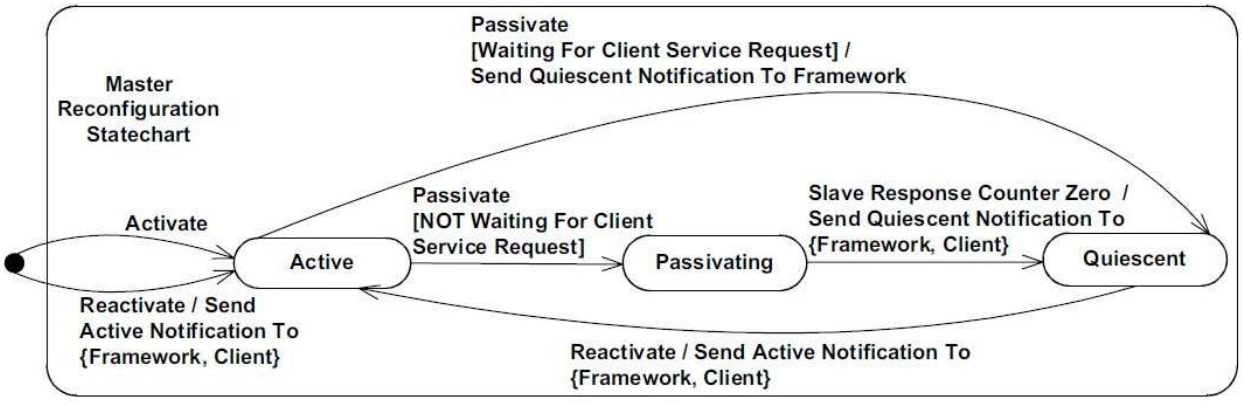
Kuvassa 20 on Master-komponentin tilakaavio kuvan 19 kuvaaman vuorovaikutuksen aikana. Master-komponentti aktivoituu, kun toiminta alkaa. Se odottaa Client-komponentilta palvelupyyntöä. Saatuaan pyynnön se lähettää viestin eteenpäin Slave-komponenteille ja jää odottamaan vastausta niiltä. Kun kaikki Slave-komponentit ovat vastanneet, se lähettää vastauksen takaisin Client-komponentille ja jää odottamaan seuraavaa palvelupyyntöä.




Kuva 20. Master-komponentin toimintaa kuvaava tilakaavio. [Gomaa and Hussein, 2004]

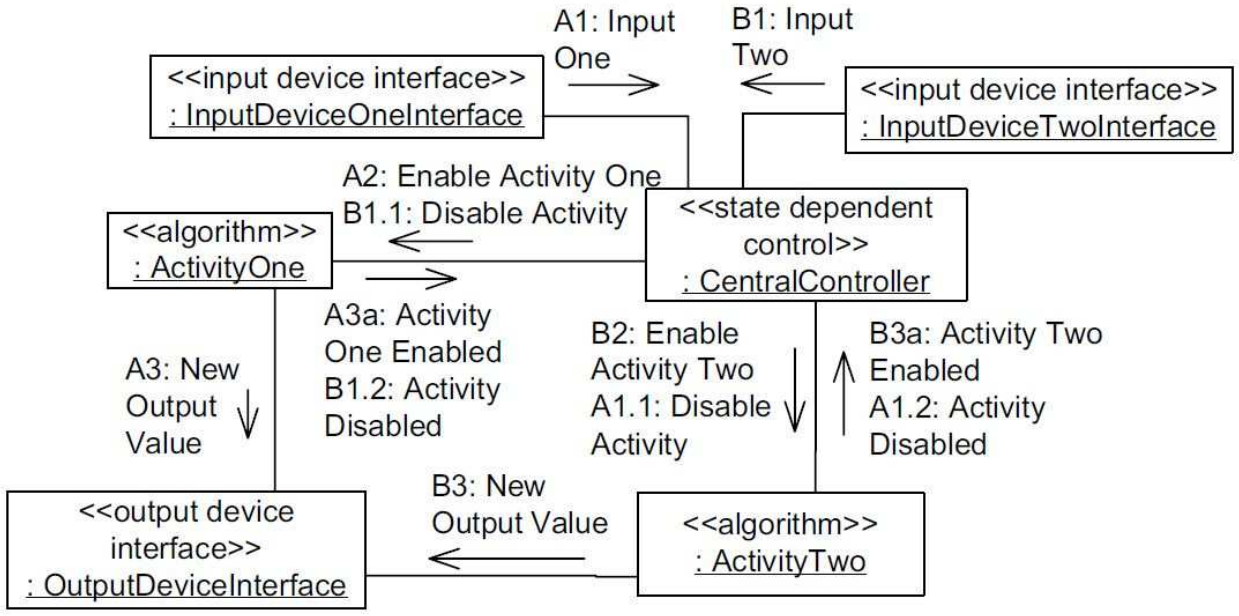
Kuvassa 21 esitetään Master-komponentin tilat uudelleenkonfiguroitumisen aikana. Mikäli komponentti ei odota Client-komponentilta palvelupyyntöä, se passivoituu, ja jos se ei odota myöskään Slave-komponenteilta vastauksia, se siirtyy toimettomaan tilaan. Mikäli se on aktiivinen ja odottaa palvelupyyntöä, se passivoituu ja lähettää järjestelmälle/kehykselle ilmoituksen siitä, että se siirtyy toimettomaan tilaan. Master-komponentti voidaan poistaa tai korvata toisella komponentilla,

jos se on toimettomassa tilassa. Myös Slave-komponentteja voidaan poistaa tai vaihtaa uusiin komponentteihin, mikäli Master-komponentti on toimettomana.



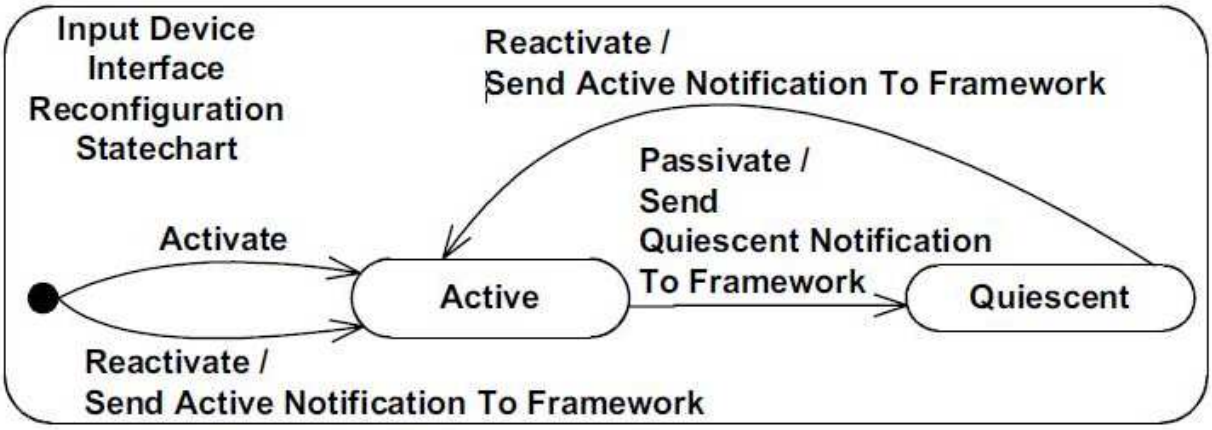

Kuva 21. Master-komponentin tilakaavio uudelleenkonfiguroitumisen aikana. [Gomaa and Hussein, 2004]

Centralized Control -mallissa on komponentti, joka ohjaa muiden komponenttien toimintaa, sekä ulkoisia laitteita, joilta järjestelmä saa syötteitä. Toiminnot riippuvat syötteistä sekä siitä, missä tilassa ohjauskomponentti on vastaanottaessaan niitä. Kuvassa 22 on Centralized Control -mallin vuorovaikutuskaavio. CentralController-komponentti saa syötteen InputDeviceOneInterface-rajapinnan kautta ja aktivoi ActivityOne-algoritmin sekä passivoi ActivityTwo-algoritmin. Jos CentralController saa syötteen InputDeviceTwoInterface-rajapinnan kautta, se aktivoi ActivityTwo-algoritmin ja passivoi ActivityOne-algoritmin. Molemmat algoritmit tuottavat jonkin tuloksen, joka välitetään OutputDeviceInterface-rajapinnan kautta esimerkiksi näytölle.



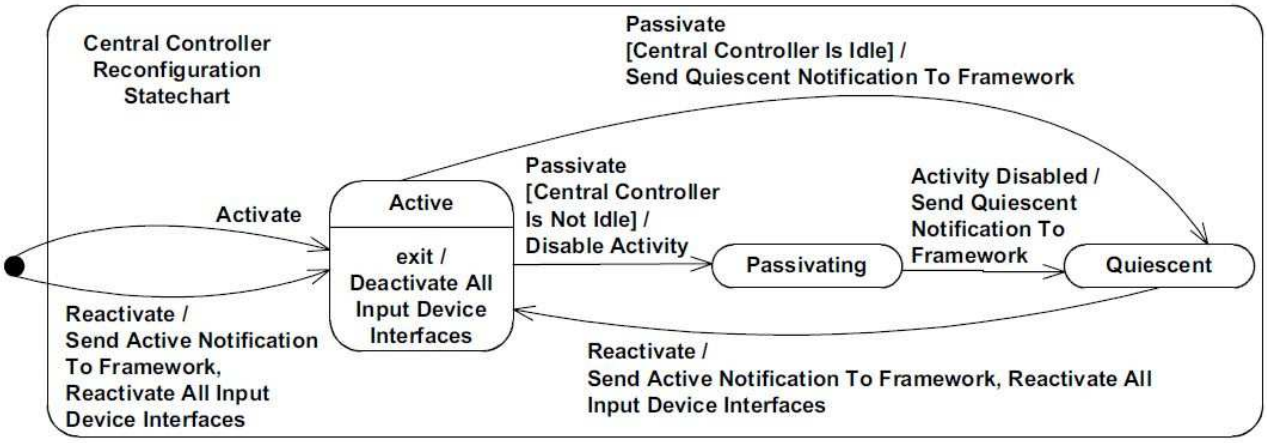

Kuva 22. Centralized Control -mallin vuorovaikutuskaavio. [Gomaa and Hussein, 2004]

Centralized Control -uudelleenkonfiguroitumismallissa CentralController-komponentin täytyy olla toimettomassa tilassa, jotta mikä tahansa komponentti voidaan poistaa tai vaihtaa toiseen. Kuvassa 23 on syötelaitteen uudelleenkonfiguroitumista kuvaava tilakaavio. Kun laite aktivoituu tai siirtyy toimettomaan tilaan, se lähettää järjestelmälle ilmoituksen siitä.




Kuva 23. Syötelaitteen tilakaavio uudelleenkonfiguroitumisessa. [Gomaa and Hussein, 2004]

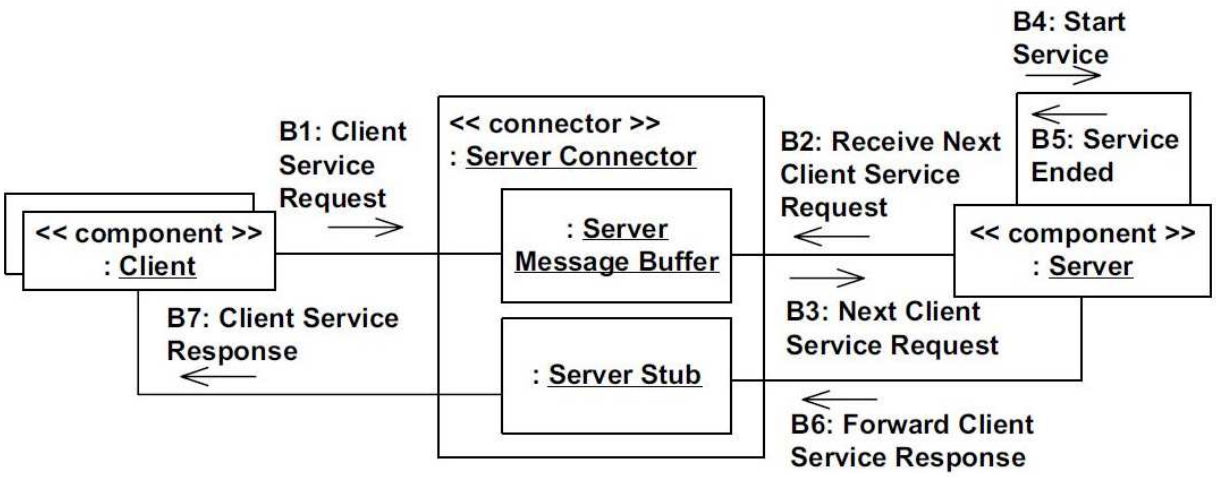
Kuva 24 esittää CentralController-komponentin tilat uudelleenkonfiguroitumisessa. Kun komponentti muuttuu aktiiviseksi, se ilmoittaa siitä järjestelmälle sekä aktivoi kaikki syötelaitteet. Aktiivisesta tilasta komponentti voi siirtyä joko passivoituvaan tilaan tai uinuvaan tilaan. Jos komponentti on toimeton, se siirtyy uinuvaan tilaan ja ilmoittaa siitä järjestelmälle. Jos taas komponentti ei ole toimeton, se poistaa toiminnot käytöstä ja siirtyy passivoituvaan tilaan. Kun toiminnot on poistettu käytöstä, se lähettää järjestelmälle ilmoituksen uinuvaan tilaan siirtymisestä.




Kuva 24. CentralController-komponentin tilakaavio uudelleenkonfiguroitumisessa. [Gomaa and Hussein, 2004]

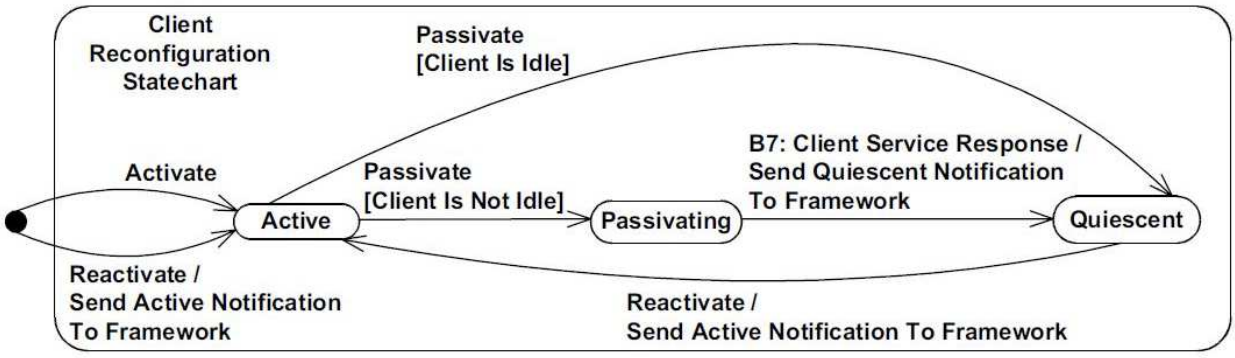
Client/Server -mallissa Client-komponentit käyttävät Server-komponenttien tarjoamia palveluita ja komponentit kommunikoivat keskenään lähettämällä toisilleen viestejä. Sekä Client- että Server-komponentteja voi olla useita. Client-komponentin poistaminen ei vaikuta järjestelmän toimin-

taan millään tavoin, vaan jäljelle jäävät Client- ja Server-komponentit voivat kommunikoida keskenään normaalisti. Sen sijaan Server-komponentin poistaminen vaikuttaa järjestelmän toimintaan, siten että sen tarjoamat palvelut eivät enää ole saatavilla. Kuvassa 25 esitetään mallin komponenttien vuorovaikutus. Client-komponentti lähettää palvelupyynnön Server-komponentille. Komponenttien välissä toimiva ServerConnector-komponentti puskuroi Client-komponenteilta tulevat pyynnöt ja välittää ne saapumisjärjestyksessä Server-komponentille. Server-komponentti suorittaa palvelun ja lähettää vastauksen Client-komponentille ServerConnector-komponentin kautta.

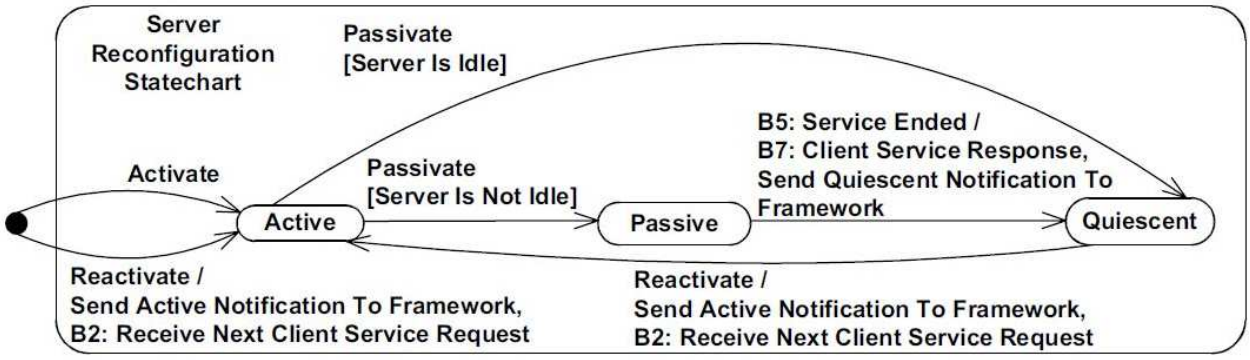



Kuva 25. Client/Server -mallin vuorovaikutuskaavio. [Gomaa and Hussein, 2004]

Client/Server -uudelleenkonfiguroitumismallissa Server-komponentti voidaan vaihtaa, kun käynnissä olevat transaktiot on suoritettu loppuun. Samoin Client-komponentti voidaan poistaa, kun sen aloittama transaktio on loppunut. Kuvassa 26 esitetään Client-komponentin tilat. Jos se on aktiivinen, se siirtyy uinuvaan tilaan, mikäli se on toimeton, ja passivoituvaan tilaan, mikäli se ei ole toimeton. Passivoituvasta tilasta se siirtyy uinuvaan tilaan, kun se saa Server-komponentilta vastauksen pyyntöönsä. Uinuvasta tilasta se voi siirtyä taas aktiiviseksi.



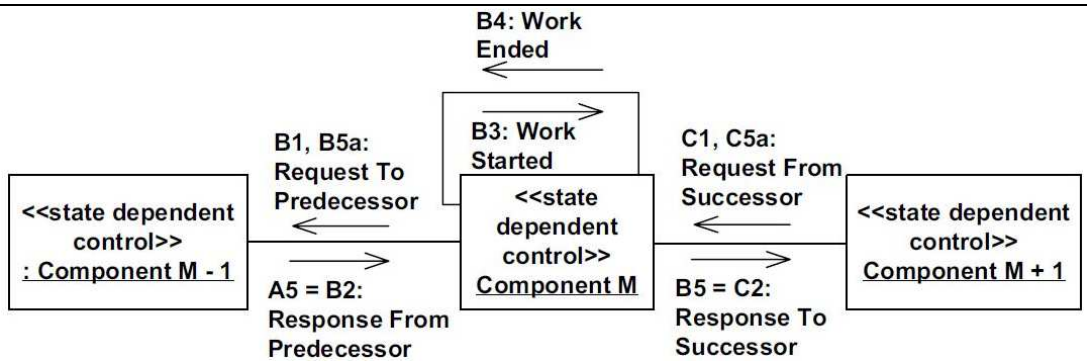

Kuva 26. Client-komponentin tilakaavio. [Gomaa and Hussein, 2004]




Kuva 27. Server-komponentin tilakaavio. [Gomaa and Hussein, 2004]

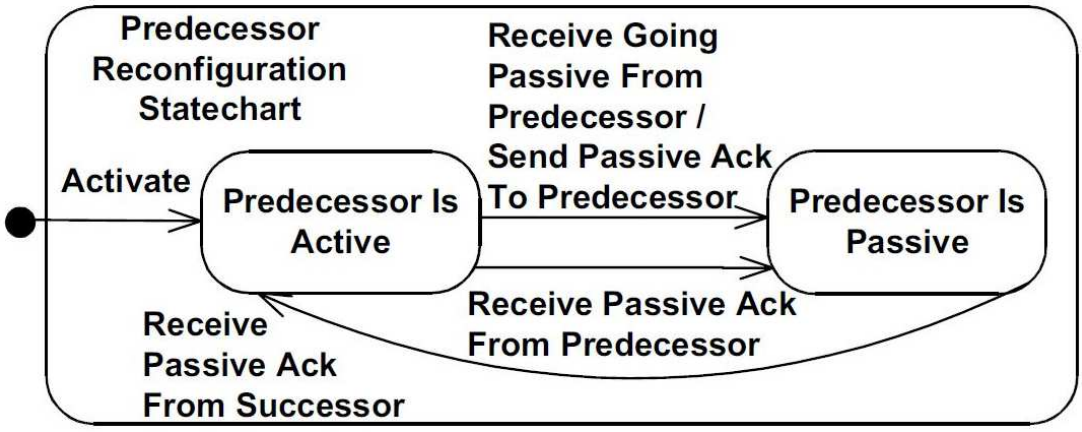
Kuvassa 27 on Server-komponentin tilakaavio. Aktiivisesta tilasta se voi siirtyä joko uinuvaan tilaan, mikäli se on toimeton, tai passiiviseen tilaan, mikäli jokin transaktio on vielä käynnissä. Passiivisesta tilasta se siirtyy uinuvaan tilaan, kun se saa suoritettua transaktion loppuun, lähettää vastauksen Client-komponentille sekä järjestelmälle ilmoituksen uinuvaan tilaan siirtymisestä. Uinuvasta tilasta se voi jälleen aktivoitua, kun se vastaanottaa seuraavan palvelupyynnön joltakin Client-komponentilta. Samalla se lähettää järjestelmälle tiedon, että on jälleen aktiivisessa tilassa.

Decentralized Control -suunnittelumallia käytetään usein hajautetuissa järjestelmissä eikä siinä ole yhtä tiettyä komponenttia, joka koordinoisi koko järjestelmän toimintoja. Sen sijaan siinä on useita ohjauskomponentteja. Mallin komponentit voivat liittyä toisiinsa monella eri tavalla, esimerkiksi renkaan muotoon tai peräkkäin. Jos komponentit muodostavat renkaan, jokaisella niistä on sekä vasen että oikea yhteys toiseen komponenttiin. Jos komponentit ovat peräkkäin, jokaisella komponentilla on samanlaiset yhteydet, joilla ne kommunikoivat sekä edeltävän että seuraavan komponentin kanssa. Kuvassa 28 on vuorovaikutuskaavio Decentralized Control -mallista, jossa on kolme komponenttia peräkkäin. Kaaviossa on kolme prosessia meneillään (A, B ja C). Prosessi A liittyy Component M-1 -komponentin toimintaan, prosessi B Component M -komponentin toimintaan ja prosessi C liittyy Component M+1 -komponentin toimintaan. Keskimmäinen komponentti lähettää edeltävälle komponentille pyynnön (B1), jonka edeltävä komponentti käsittelee (A3 ja A4, jotka eivät näy kuvassa) ja lähettää vastauksen (B2) keskimmäiselle komponentille. Keskimmäinen komponentti suorittaa jonkin toiminnon (B3 ja B4), minkä jälkeen se lähettää vastauksen seuraavalle komponentille (B5 = C2).



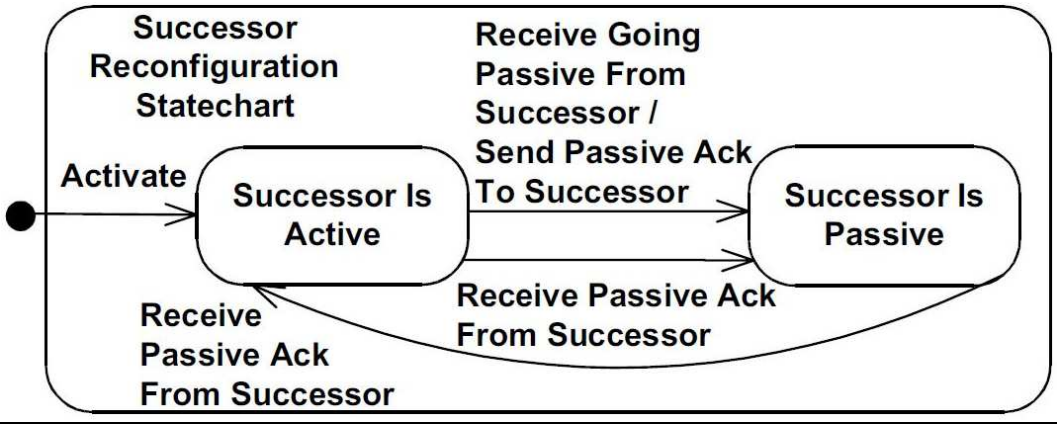

Kuva 28. Decentralized Control -mallin vuorovaikutuskaavio. [Gomaa and Hussein, 2004]

Decentralized Control -uudelleenkonfiguroitumismallissa komponentti voi lakata kommunikoimasta viereisen komponentin kanssa, mutta jatkaa muuta toimintaansa. Jotta komponentti voidaan poistaa tai vaihtaa, yhteydet sen ja sen viereisten komponenttien väliltä täytyy katkaista. Komponentin on aina oltava tietoinen edeltävän ja seuraavan komponentin tilasta. Kuva 29 kuvaa sitä, miten komponentti tuntee edeltäjänsä tilat. Sillä on kaksi tilaa, Edeltäjä on aktiivinen ja Edeltäjä on passiivinen. Kun komponentti saa edeltäjältään ilmoituksen siitä, että edeltäjä siirtyy passiiviseen tilaan, se lähettää edeltäjälle kuittauksen. Tällöin komponentti siirtyy Edeltäjä on aktiivinen -tilasta Edeltäjä on passiivinen -tilaan. Komponentti siirtyy tilan myös silloin, kun se saa edeltäjältä ilmoituksen, että se on passiivinen. Takaisin Edeltäjä on aktiivinen  -tilaan komponentti siirtyy, kun se saa seuraajakomponentilta ilmoituksen, että seuraaja on passiivinen.

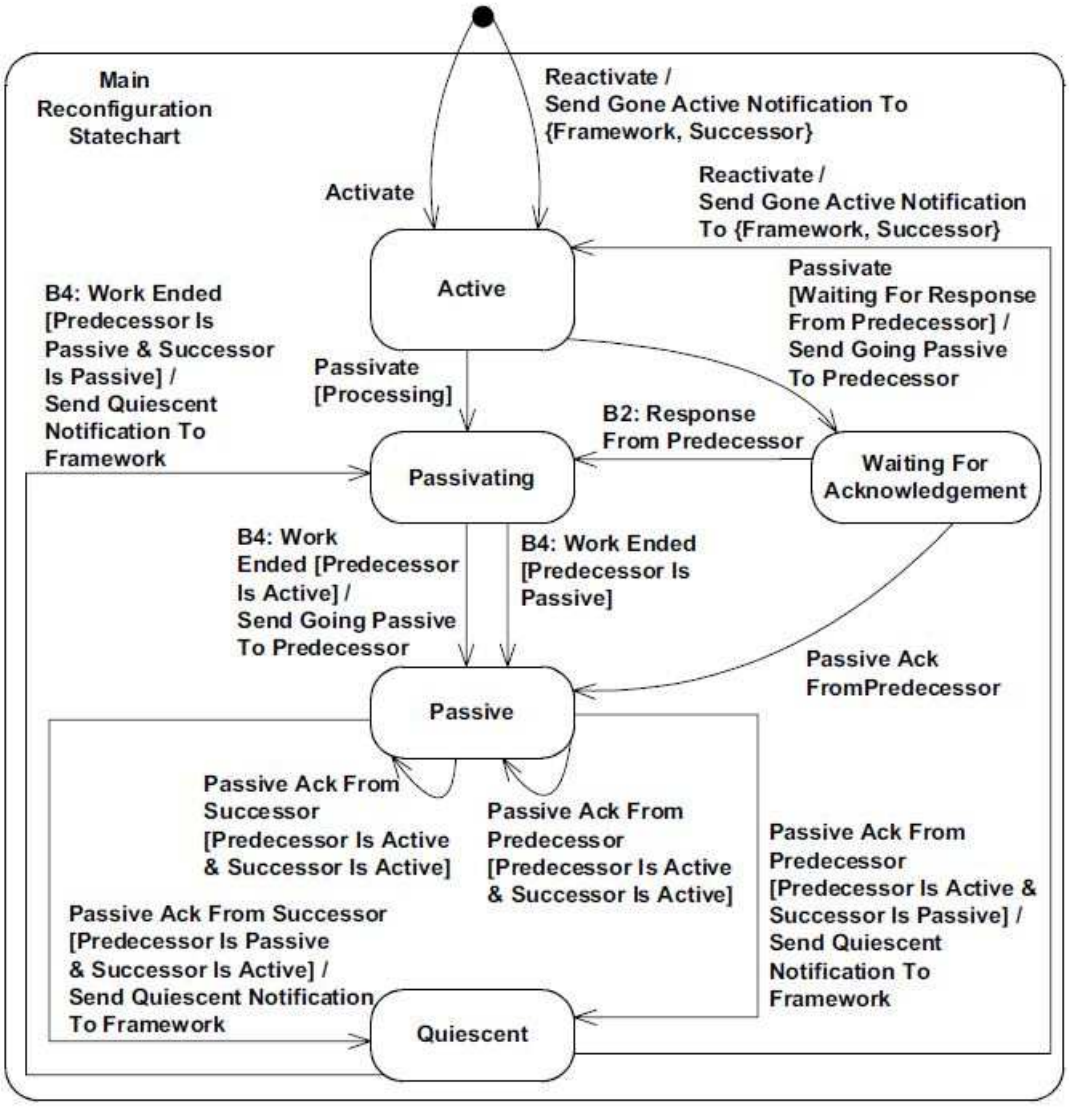



Kuva 29. Komponentin tilakaavio. [Gomaa and Hussein, 2004]

Kuvassa 30 esitetään ne tilat, joilla kuvataan komponentin tietoisuutta Seuraaja-komponentin tilasta. Kun komponentti saa seuraajaltaan viestin, jossa se kertoo muuttuvansa passiiviseksi, komponentti siirtyy Seuraaja on passiivinen -tilaan. Samoin jos se lähettää seuraajalleen käskyn passivoitua tai saa seuraajalta ilmoituksen passivoitumisesta, se siirtyy Seuraaja on passiivinen -tilaan. Takaisin Seuraaja on aktiivinen -tilaan komponentti siirtyy, mikäli se Seuraaja on passiivinen -tilassa ja saa seuraajalta ilmoituksen passivoitumisesta.




Kuva 30. Seuraaja-komponentin tilakaavio. [Gomaa and Hussein, 2004]




Kuva 31. Komponentin tilakaavio. [Gomaa and Hussein, 2004]

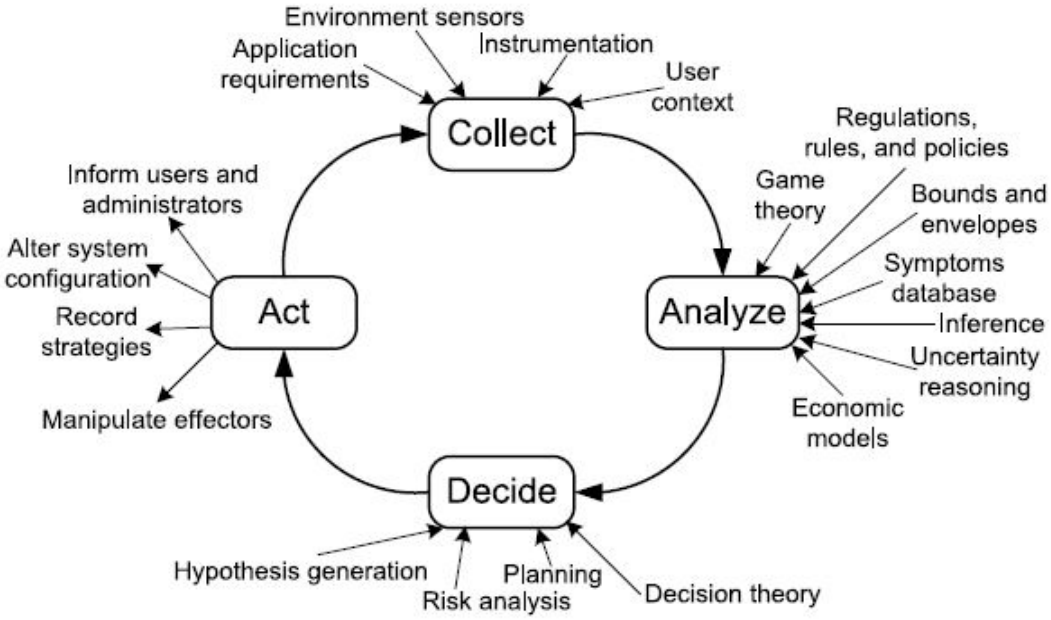
Kuvassa 31 esitetään komponentin tilat ja niiden väliset siirtymät sekä siirtymien vaatimat ehdot. Komponentilla on tilat Aktiivinen, Passivoituva, Passiivinen, Uinuva ja Odottaa kuittausta. Komponentti siirtyy Aktiivisesta Passivoituvaksi, kun sillä on prosessointi kesken ja siitä edelleen Passiiviseksi, kun prosessointi päättyy ja kun se on Edeltäjä on aktiivinen -tilassa, tai kun se lähettää edeltäjälle ilmoituksen siirtymisestä passiiviseksi, tai kun prosessointi päättyy ja komponentti on Edeltäjä on passiivinen -tilassa. Komponentin ollessa Passiivinen edeltäjä ja seuraaja voivat lähettää sille kuittauksen siitä, että ne ovat passiivisia. Jos komponentti on passiivinen ja sen edeltäjä ja seuraaja ovat aktiivisia ja komponentti saa jommaltakummalta vierekkäisistä komponenteista kuittauksen siitä, että kuittauksen lähettäjä on passiivinen, komponentti pysyy passiivisena. Jos taas edeltäjä on passiivinen ja seuraaja on aktiivinen ja komponentti saa seuraajalta kuittauksen passiiviseksi siirtymisestä, se ilmoittaa järjestelmälle siirtyvänsä uinuvaan tilaan. Samoin jos komponentin ollessa passiivinen ja sen seuraaja on passiivinen ja edeltäjä aktiivinen ja edeltäjä siirtyy passiiviseen tilaan, komponentti siirtyy uinuvaan tilaan ja ilmoittaa siitä järjestelmälle. Uinuvasta tilasta komponentti voi siirtyä aktiiviseen tilaan, mistä se ilmoittaa sekä seuraajalle että järjestelmälle, tai

passivoituvaan tilaan, mikäli prosessointi loppuu ja molemmat viereiset komponentit ovat passiivisia. Aktiivisesta tilasta komponentti voi siirtyä odottamaan kuittausta, mikäli se odottaa vastausta edeltäjältään. Tällöin se ilmoittaa edeltäjälleen passivoituvansa. Kun se saa edeltäjältä vastauksen, se siirtyy passivoituvaan tilaan.

# 6. Ohjaussilmukat

## 6.1. Yleistä ohjaussilmukoista

Itseohjautuvien järjestelmien perusominaisuus itseohjautuvuus toteutetaan useimmiten jonkinlaisen ohjaussilmukan avulla. Yksi tapa toteuttaa tällainen ohjaussilmukka on MAPE-silmukka. MAPE-silmukassa on neljä komponenttia, joilla jokaisella on oma tehtävänsä. Komponenttien tehtävät ovat tarkkaileminen (monitoring), analysointi (analyzing), suunnittelu (planning) ja toimeenpano (execution). Tarkkailijakomponentti seuraa järjestelmän suoritusta ja järjestelmässä sekä sen ympäristössä tapahtuvia muutoksia. Analysointikomponentin tehtävä on analysoida, onko muutokseen reagoitava jollakin tavalla. Mikäli analyysin tuloksena järjestelmän täytyy mukautua, suunnittelijakomponentti tekee päätöksen siitä, miten järjestelmä muuttaa itseään. Toimeenpanokomponentin tehtävänä on laittaa muutos alulle. Kuvassa 32 esitetään ohjaussilmukan komponenttien toiminnot.




Kuva 32. Ohjaussilmukan toiminnot. [Cheng et al., 2009]

Weyns ja muut [2012] esittelevät malleja, joilla voidaan toteuttaa itseohjautuvien järjestelmien ohjaussilmukoita. He käsittelevät sellaisia malleja, joissa ohjaustoiminnallisuus ei ole keskitettyä eli niissä on useita ohjaussilmukoita. Näitä malleja he kutsuvat nimillä Coordinated Control, Information Sharing, Master-Slave, Regional Planning ja Hierarchical Control. Näistä kahdessa ensimmäisessä käytetään rinnakkaisia ohjaussilmukoita, kun taas loput kolme perustuvat siihen, että korkeamman tason ohjaussilmukat kontrolloivat alemman tason ohjaussilmukoita.

## 6.2. Rinnakkaiset ohjaussilmukat

Coordinated Control -mallia voidaan käyttää silloin, kun mukautumiseen tarvittava tieto on hajanaisesti järjestelmässä, järjestelmä on mittakaavaltaan niin suuri, että tiedon käsittely keskitetysti ei

kannata, tai kun järjestelmä ulottuu monelle sellaiselle alueelle, joilla ei ole luotettavaa tahoa kontrolloimassa mukautumista. Tällaisissa tilanteissa jokaisella rinnakkaisella järjestelmän osalla on oma ohjaussilmukkansa, joiden MAPE-komponentit kommunikoivat rinnakkaisten silmukoiden vastaavien komponenttien kanssa. Coordinated Control -mallin etuna on sen skaalautuvuus. Lisäksi se voi tehdä järjestelmästä vakaamman, sillä yhden osakomponentin kaatuminen ei kaada koko järjestelmää. Ongelmaksi voi sen sijaan muodostua se, että ei voida taata kaikkien ohjaussilmukoiden toteuttavan oman ohjattavan järjestelmänsä mukautumista yhdenmukaisesti. Lisäksi rinnakkaisten ohjaussilmukoiden yhdessä tekemät mukautumispäätökset saattavat johtaa siihen, että järjestelmä kokonaisuutena ei enää toimi optimaalisella tavalla. [Weyns et al., 2012]

Information Sharing -malli on hyödyllinen silloin, kun järjestelmän komponentit ovat löyhästi yhteydessä toisiinsa ja kun järjestelmän osat voivat mukautua paikallisesti, mutta tarvitsevat tietoa muiden osien tilasta, koska paikalliset muutokset voivat vaikuttaa muihinkin osiin. Tässä mallissa ainoastaan rinnakkaisten ohjaussilmukoiden tarkkailijakomponentit kommunikoivat keskenään. Muut komponentit kommunikoivat vain saman silmukan muiden komponenttien kanssa. Information Sharing -mallin etuja ovat sen skaalautuvuus ja ajantasaisuus. Koska analyysi-, suunnittelu- ja toimeenpanokomponentit eivät kommunikoi ulkopuolisten komponenttien kanssa, järjestelmä kykenee tekemään päätökset ja käynnistämään mukautumismekanismin ajantasaisesti. Huonona puolena voidaan pitää sitä, että paikalliset mukautumiset voivat vaikuttaa negatiivisesti globaalissa mittakaavassa. Paikallisesti tehdyt päätökset voivat myös olla ristiriidassa toistensa kanssa, mikä johtaa pysyviin muutoksiin järjestelmässä. Tämä voi vaikuttaa resurssien käyttöön sekä järjestelmän saavutettavuuteen ja vakauteen. [Weyns et al., 2012]

### 6.3. Hierarkkiset ohjaussilmukat

Master-Slave -mallissa on yksi suunnittelijakomponentti ja yksi analyysikomponentti. Tarkkailija- ja toimeenpanokomponentteja voi olla useampia. Jokainen tarkkailijakomponentti kommunikoi analyysikomponentin kanssa ja jokainen toimeenpanokomponentti suunnittelijakomponentin kanssa. Analyysi- ja suunnittelijakomponentit kommunikoivat keskenään. Tätä mallia voidaan käyttää, kun tiedon kerääminen ja mukautuminen täytyy tapahtua paikallisesti, mutta paikalliset muutokset vaikuttavat myös muiden osajärjestelmien toimintaan. Mallin etuna on se, että analysoinnin ja suunnittelun keskittämisen avulla globaalien tavoitteiden saavuttaminen on tehokkaampaa. Keskittämisestä voi kuitenkin olla myös haittaa, sillä varsinkin suuremman mittakaavan hajautetussa järjestelmässä tiedon prosessointi voi olla hidasta. [Weyns et al., 2012]

Regional Planning -mallia voidaan käyttää silloin, kun järjestelmässä on useita löyhästi toisistaan riippuvaisia osia, jotka vaativat paikallista mukautumista mutta myös osien välistä mukautumista. Jokaisella osalla on yksi suunnittelijakomponentti, joka suunnittelee paikallisen mukautumisen ja joka kommunikoi muiden osien suunnittelijakomponenttien kanssa suunnitellakseen osien välisen mukautumisen. Yhdessä osassa on yksi tai useampia tarkkailija-, analyysi- ja toimeenpanokomponentteja. Tarkkailijat keräävät tietoa osan alijärjestelmistä, analyysikomponentit analysoivat

tiedon ja raportoivat sen suunnittelijakomponentille, joka yksin tai yhdessä muiden suunnittelijoiden kanssa päättää, miten järjestelmän täytyy mukautua, ja välittää tiedon tarvittaville toimeenpanokomponenteille. Regional Planning -mallin etuna on se, että sen avulla yhden osan sisällä eri ohjaussilmukat voivat keskittyä eri asioihin, mutta suunnittelu tapahtuu kuitenkin ylemmällä tasolla. Lisäksi paikallinen tarkkailu ja analyysi voivat vähentää vuorovaikutusta suunnittelijakomponentin kanssa, mikä parantaa suorituskykyä. Riskinä on kuitenkin, että mukautuminen ei ole tehokasta. Lisäksi mukautumisen toimeenpano täytyy suunnitella huolellisesti, sillä malli ei tue toimeenpanokomponenttien välistä vuorovaikutusta. [Weyns et al., 2012]

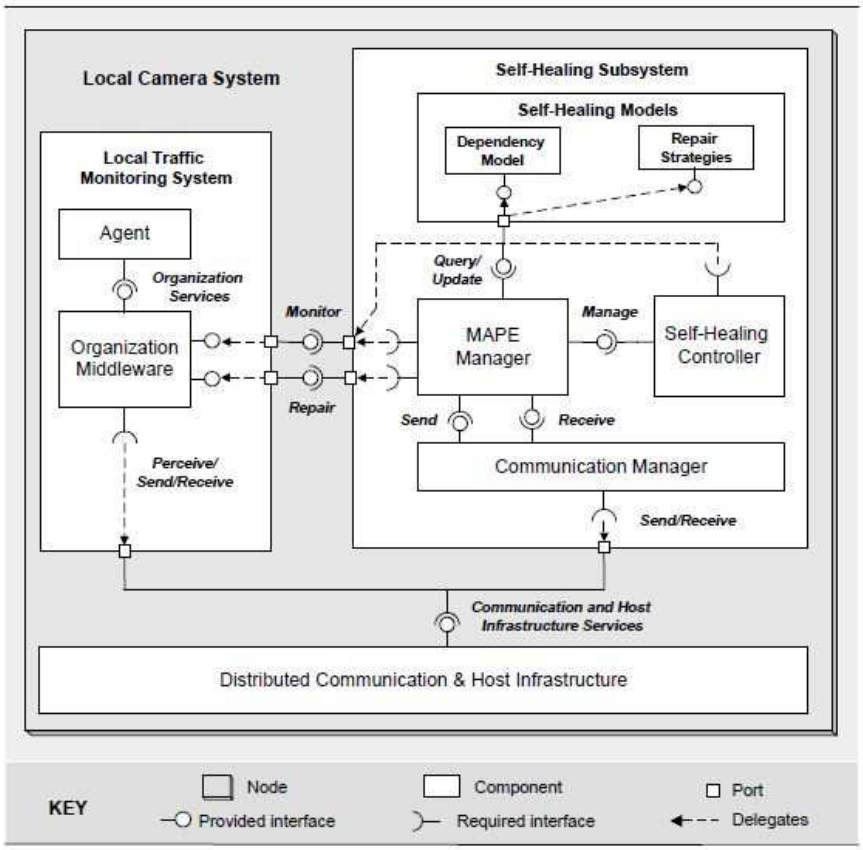
Hierarchical Control -mallia voidaan käyttää silloin, kun ohjaava alijärjestelmä on monimutkainen ja sen itsensä täytyy mukautua. Siinä on useita ohjaussilmukoita, jotka hallitsevat eri resursseja tai joiden toiminta-aika on erilainen. Niiden täytyy kuitenkin kommunikoida keskenään välttääkseen ristiriitoja. Hierarchical Control -mallissa ohjaussilmukat ovat hierarkkisessa suhteessa toisiinsa. Eri kerrosten silmukat hallitsevat eri asioita eri abstraktiotasoilla. Alimpien kerrosten silmukat toimivat lyhyellä aikavälillä ja ylimmän kerroksen silmukat pitkällä aikavälillä. Alimman kerroksen silmukat hallitsevat ohjattavaa alijärjestelmää ja välikerrosten silmukat alempia kerroksia. Kaikkein ylimmällä tasolla oleva silmukka on vastuussa koko järjestelmän ohjaamisesta. Alimman tason silmukoiden tarkkailija- ja toimeenpanokomponentit kommunikoivat ohjattavan alijärjestelmän kanssa, ja sitä ylempien tasojen vastaavat silmukat ovat vuorovaikutuksessa niiden alapuolella olevan kerroksen silmukoiden kanssa. Kaikkien tasojen analyysi- ja suunnittelijakomponentit ovat vuorovaikutuksessa ainoastaan oman silmukkansa muiden komponenttien kanssa. [Weyns et al., 2012]

# 7. Liikenteenvalvontajärjestelmän arkkitehtuuri

Kohdassa 2.1. annoin muutaman esimerkin järjestelmistä, joissa on itseohjautuvaa toiminnallisuutta. Yksi esimerkeistä oli liikenteenvalvontajärjestelmä, jonka arkkitehtuuriin syvennyn tarkemmin tässä luvussa.

Liikenteenvalvontajärjestelmän tarkoituksena on tarkkailla liikennettä ruuhkien varalta. Järjestelmän keräämää tietoa voivat käyttää hyväkseen esimerkiksi liikennevalojenohjausjärjestelmät siten, että ne pyrkivät purkamaan ruuhkia ajoittamalla vihreät valot sopivasti, tai navigointijärjestelmät opastamalla kuljettajat ajamaan ruuhkattomia tieosuuksia. Liikenteenvalvontajärjestelmä koostuu itsenäisistä älykkäistä kameroista, jotka on sijoitettu tieosuuksille tasaisin välimatkoin. Jokainen kamera pystyy valvomaan rajoitettua aluetta, ja järjestelmän itseohjautuvana ominaisuutena on itsekorjautuvuus. Jos joku järjestelmän kameroista hajoaa eikä enää välitä tietoa oikein, järjestelmä pystyy korjaamaan itsensä siten, että se toimii edelleen johdonmukaisesti, vaikka rikkoutuneen kameran valvomalta tieosuudelta ei saadakaan tarkkaa tietoa.

Kuvassa 33 on kameran ohjelmiston komponentit. Jokaisen kameran paikallinen järjestelmä koostuu paikallisesta liikenteenvalvontajärjestelmästä (local traffic monitoring system), itsekorjautuvasta alijärjestelmästä (self-healing subsystem) ja hajautetusta viestintä- ja isäntäinfrastruktuurista (distributed communication and host infrastructure).

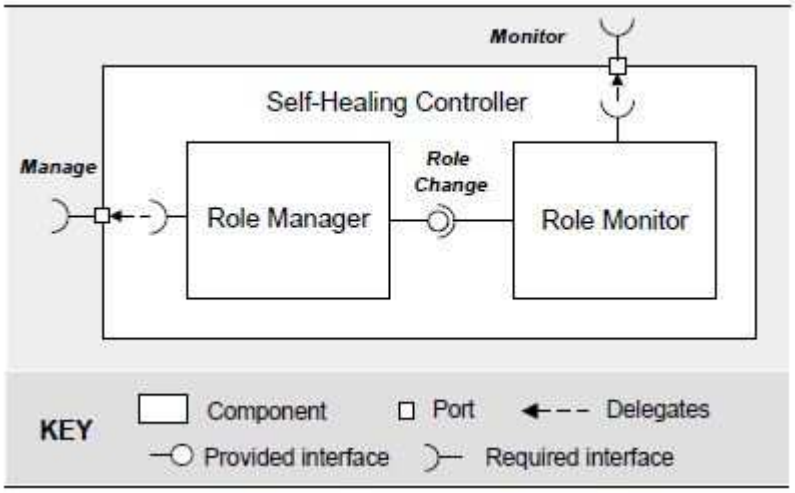



Kuva 33. Kameran komponentit. [Vromant et al., 2011].

Paikallisen liikenteenvalvontajärjestelmän tehtävänä on havaita liikenneruuhkat ja tiedottaa niistä järjestelmän asiakkaille. Se koostuu kahdesta komponentista, agentista (agent) ja organisaation väliohjelmistosta (organization middleware). Agentti vastaa liikenteen tarkkailemisesta ja vuorovaikutuksessa muiden agenttien kanssa tiedottaa ruuhkista niille tahoille, jotka ovat kiinnostuneita niistä. Kun ruuhkaa ei ole, jokaisen kameran agentti muodostaa yksijäsenisen organisaation. Kun syntyy ruuhka, joka ulottuu useamman kameran valvomalle alueelle, niiden agentit liittyvät yhteen muodostaen yhden isomman organisaation, jossa on Master-Slave -rakenne. Yksi agenteista toimii Master-komponenttina ja on siten vastuussa organisaation dynaamisuuden hallinnasta muodostaen ja hajauttaen organisaation tarpeen vaatiessa. Se on synkronoidussa tilassa sekä oman organisaationsa Slave-komponenttien että vierekkäisten organisaatioiden Master-komponenttien kanssa. Organisaation väliohjelmisto tarjoaa sellaisia palveluja, joiden avulla agentit voivat muodostaa organisaatioita.

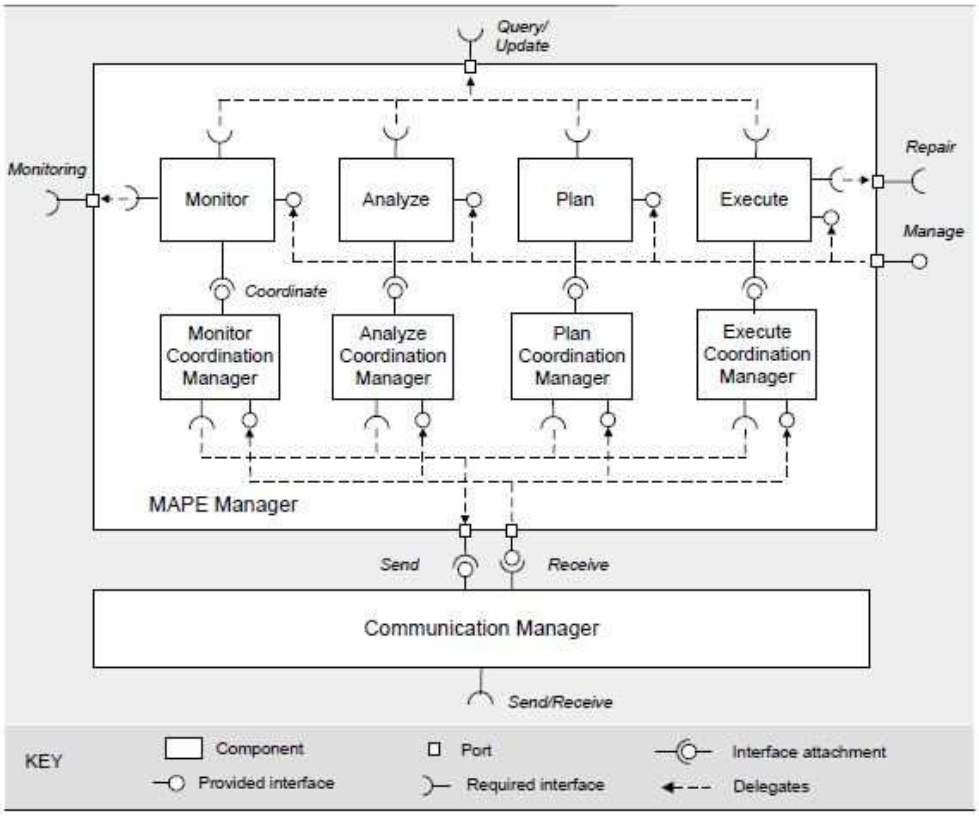
Itsekorjautuva alijärjestelmä ylläpitää itsekorjautumismalleja (self-healing models). Itsekorjautumismallit sisältävät riippuvuusmallin (dependency model), joka pitää sisällään tiedon naapuriorganisaatioista ja Master- ja Slave-komponenttien suhteista. Lisäksi itsekorjautumismalliin sisältyy joukko korjautumisstrategioita (repair strategies), joista järjestelmä valitsee kulloiseenkin tilanteeseen sopivimman. Jokaiselle ongelmatilanneskenaariolle on olemassa oma korjautumisstrategia, joka määrittelee toimenpiteet järjestelmän johdonmukaisen toiminnan takaamiseksi. Itsekorjautuva alijärjestelmä sisältää myös MAPE-ohjauskomponentin (MAPE manager), itsekorjautumiskontrollerin (self-healing controller) ja viestintäohjaimen (communication manager).

Itsekorjautumiskontrollerin tehtävänä on varmistaa, että järjestelmä tunnistaa ongelmatilanteen oikein. Sen tulee tietää, mikä on paikallisen liikenteenvalvontajärjestelmän rooli, ja sen se saa selville Monitor-rajapinnan avulla. Paikallisella liikenteenvalvontajärjestelmällä voi olla kolme roolia: yksijäsenisen organisaation Master-komponentti, monijäsenisen organisaation Master-komponentti tai monijäsenisen organisaation Slave-komponentti. Itsekorjautumiskontrollerin rakenne esitetään tarkemmin kuvassa 34. Se koostuu kahdesta komponentista, roolienhallintakomponentista (role manager) ja roolientarkkailukomponentista (role monitor). Monitor-rajapinnan kautta roolientarkkailukomponentti tarkkailee paikallisen liikenteenvalvontajärjestelmän senhetkistä roolia. Kun rooli muuttuu, roolientarkkailukomponentti lähettää siitä tiedon roolienhallintakomponentille sen tarjoaman RoleChange-rajapinnan kautta. Roolienhallintakomponentti ohjeistaa MAPE-ohjauskomponenttia toimimaan sopivalla tavalla.



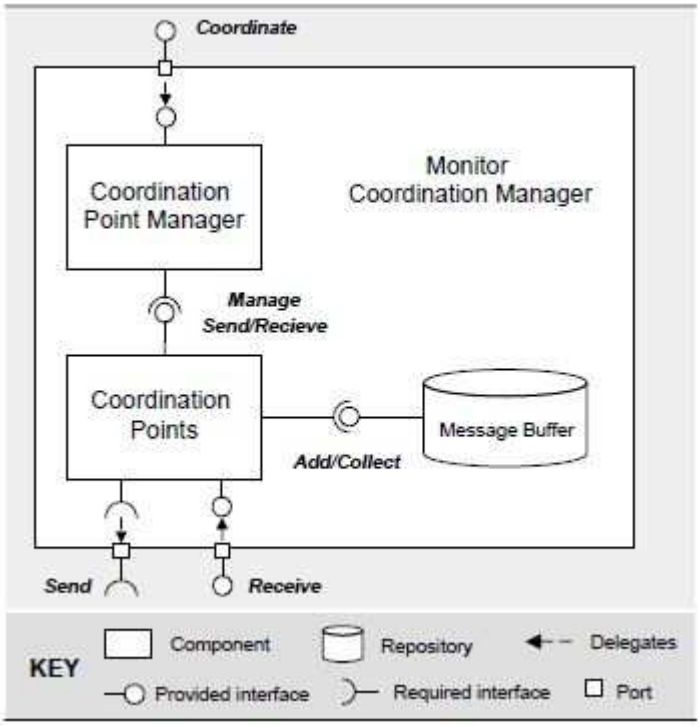

Kuva 34. Itsekorjautumiskontrollerin sisäinen rakenne. [Vromant et al. 2011]

MAPE-ohjauskomponentin tehtävänä on huolehtia itsekorjautumisesta. Se tarkkailee pääjärjestelmää Monitor-rajapinnan kautta ja ylläpitää riippuvuusmallia. Havaitakseen virhetilanteita se kommunikoi Send- ja Receive-rajapintojen avulla niiden kameroiden itsekorjautuvien alijärjestelmien kanssa, joiden kanssa sillä on riippuvuuksia. Kuvassa 35 on MAPE-ohjauskomponentin rakenne. Se koostuu kahdeksasta osakomponentista, tarkkailija-, analyysi-, suunnittelu- ja toimeenpanokomponentista, joiden tehtävät on kuvattu luvussa 6, sekä niiden jokaisen koordinaatiokomponenteista.




Kuva 35. MAPE-ohjauskomponentin rakenne. [Vromant et al., 2011]

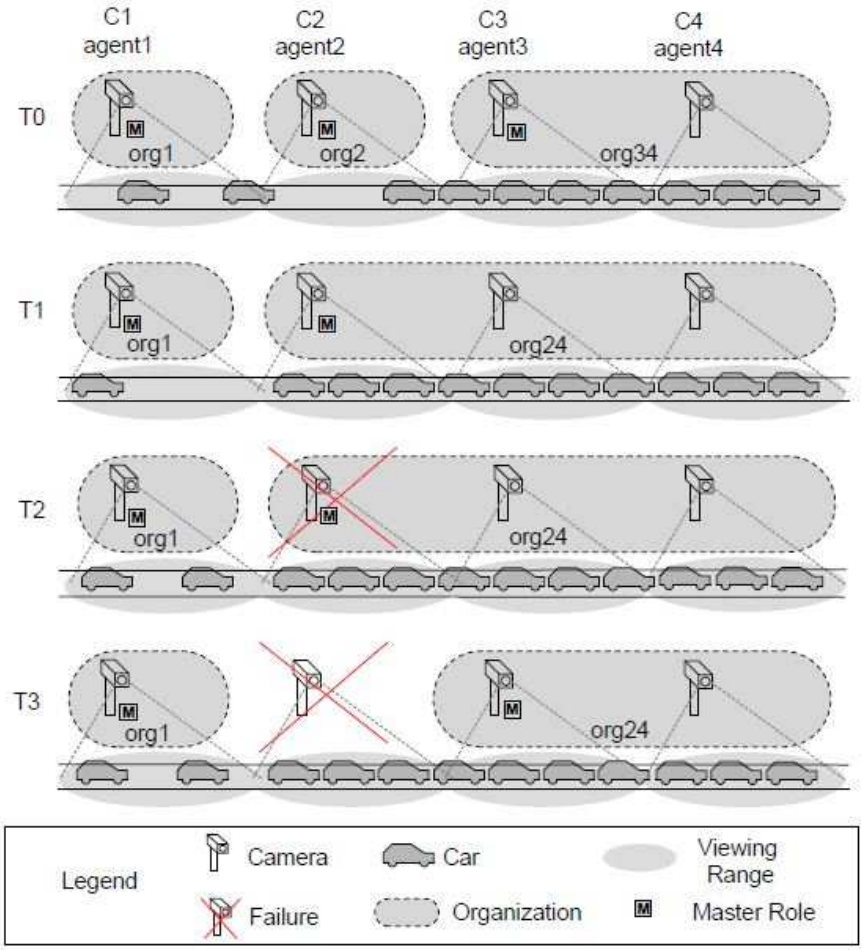
Kuvassa 36 on MAPE-ohjauskomponentin sisäisen tarkkailijakomponentin koordinaatiokomponentti. Sen tehtävänä on auttaa tarkkailijakomponenttia toimimaan asynkronisesti yhdessä muiden MAPE-komponenttien kanssa niiden toteuttaessa itsekorjautumisskenaarioita. Sillä on kolme osakomponenttia, koordinaatiopisteohjain (coordination point manager), koordinaatiopisteet (coordination points) ja viestipuskuri (message buffer). Tarkkailijakomponentti käyttää koordinaatiopisteohjaimen tarjoamaa Coordinate-rajapintaa rekisteröidäkseen uusia itsekorjautumisskenaarioita muiden komponenttien kanssa. Skenaarion rekisteröiminen luo uuden koordinaatiopisteen, joka lisätään koordinaatiopisteiden joukkoon. Koordinaatiopisteen tehtävänä on hallita tarkkailijakomponentin ja koordinaatiopistettä vastaavan skenaarion yhteistoimintaa. Se lähettää muille saman MAPE-ohjauskomponentin osakomponenteille viestejä viestintäohjaimen tarjoaman Send-rajapinnan kautta, ja viestintäohjain välittää ne oikealle koordinaatio-ohjaimelle Receive-rajapinnan kautta. Koordinaatiopiste puskuroi saapuvat viestit viestipuskuriin odottamaan käsittelyä. Tarkkailijakomponentti voi pyytää skenaarioon liittyviä viestejä koordinaatio-ohjaimelta kutsumalla Coordinate-rajapinnan Receive-operaatiota. Myös analyysi-, suunnittelu- ja toimeenpanokomponenttien ja niitä vastaavien koordinaatio-ohjainten vuorovaikutus tapahtuu samalla tavalla.




Kuva 36. Koordinaatiokomponentin rakenne. [Vromant et al., 2011]

Viestintäohjaimen tehtävänä on helpottaa sekä ohjaussilmukoiden sisäistä viestintää että eri kameroiden MAPE-ohjauskomponenttien välistä viestintää. Se tarjoaa Send-rajapinnan, jonka kautta MAPE-ohjauskomponentti voi lähettää viestejä muiden kameroiden vastaaville komponenteille. Nämä viestit välitetään Send/Receive-rajapinnan kautta hajautetulle viestintäinfrastruktuurille, josta vastaanottajat pääsevät niihin käsiksi.

Kuvassa 37 esitetään skenaario, jossa yksi kamera rikkoutuu. Alkutilanteessa (T0) on kaksi yksijäsenistä organisaatiota (org1 ja org2) sekä yksi kaksijäseninen organisaatio (org34). Kun liikenne ruuhkautuu myös kameran C2 valvomalla alueella (T1), organisaatiot org2 ja org34 sulautuvat yhteen organisaatioksi org24, jonka Master-komponentiksi tulee C2:n agentti. Kun C2 vikaantuu (T2), ne komponentit, jotka ovat vuorovaikutuksessa sen kanssa, eli tässä tapauksessa org24-organisaation Slave-komponentit (C3 ja C4) sekä naapuriorganisaatioiden Master-komponentit (C1), havaitsevat ongelman ja korjaavat sen. C3 valitaan org24:n uudeksi Master-komponentiksi (T3) ja C2 ja C3 päivittävät ajan tasalle tiedot naapuriorganisaatioistaan.




Kuva 37. Skenaario liikenteenvalvontajärjestelmän itsekorjautumisesta. [Vromant et al., 2011]

# 8. Yhteenveto

Tarve järjestelmille, jotka kykenevät mukautumaan niissä itsessään tai toimintaympäristössä tapahtuvien muutosten mukaan, on kasvamassa, sillä järjestelmien ylläpito vie paljon resursseja ja aiheuttaa lisäkustannuksia. Ylläpitokustannuksia voidaan vähentää, jos järjestelmä pystyy itse optimoimaan suorituskykyään, korjaamaan virheitä tai muulla tavoin ylläpitämään toimintakykyään.

Käsittelin tutkielmassa itseohjautuvien järjestelmien suunnittelua ja rakennetta. Itseohjautuvien järjestelmien suunnitteluun vaikuttavat järjestelmän tavoitteet, mukautumisen taustalla olevat muutokset, mekanismit, joiden avulla järjestelmän mukautuminen toteutetaan sekä mukautumisen vaikutukset. Kartoitin tutkielmassa, millaisilla suunnitteluratkaisuilla järjestelmiin voidaan toteuttaa itseohjautuvia ominaisuuksia.

Erilaisten itseohjautuvien ominaisuuksien toteuttamiseen on jo olemassa monenlaisia ratkaisuja, joita tässä tutkielmassa on esitelty. Useissa arkkitehtuurityyleissä on piirteitä, joita voidaan käyttää pohjana järjestelmien itseohjautuvuuden toteuttamisessa, ja arkkitehtuurityyli pitäisikin valita sen mukaan, mitä itseohjautuvia ominaisuuksia järjestelmään halutaan. Tärkeä osa itseohjautuvia järjestelmiä ovat ohjaussilmukat, joiden tarkkailijakomponentit keräävät tietoa järjestelmästä ja sen ympäristöstä. Erityisesti tarkkailemiseen on olemassa erilaisia malleja, joista ohjelmistojen suunnittelijat voivat valita kulloiseenkin järjestelmään parhaiten sopivan. Jotkut mallit sopivat järjestelmän itsensä tarkkailemiseen, kun taas toiset mallit ovat sopivia ympäristön havainnointiin.

Itseohjautuvuudessa olennaista on järjestelmän rakenteen muuttaminen. Myös siihen on olemassa useita erilaisia malleja. Keskeistä rakenteen muuttamisessa on komponenttien lisääminen ja poistaminen. Uusien komponenttien lisäys on tarpeen silloin, kun halutaan jakaa järjestelmän kuormitusta useammalle komponentille. Tällöin saadaan pidettyä suorituskyky samalla tasolla. Komponenttien poistaminen voi olla tarpeellista, jos järjestelmä vikaantuu. Silloin virheellinen komponentti voidaan eristää muista komponenteista ja näin estää sitä aiheuttamasta lisävahinkoa järjestelmän toiminnalle. Komponenttien lisäämisessä ja poistamisessa on kuitenkin tärkeää huolehtia siitä, että niiden komponenttien, jotka toimivat vuorovaikutuksessa lisättävien tai poistettavien komponenttien kanssa, toiminta ei saa häiriintyä järjestelmän rakenteen muutoksesta. Toisin sanoen niillä ei saisi olla mitään tehtäviä kesken, kun komponentti poistetaan tai lisätään, sillä se voi aiheuttaa epäjohdonmukaista käyttäytymistä järjestelmässä.

Järjestelmän rakenteen muuttamisen lisäksi itseohjautuvuus voi tarkoittaa myös järjestelmän käyttäytymisen muuttamista. Eri olosuhteissa järjestelmän toiminta voi olla erilaista. Tällöin järjestelmän tulee sisältää jonkinlainen malli siitä, missä olosuhteissa sen tulee toimia milläkin tavalla. Eräs ratkaisu tällaisen toiminnallisuuden toteuttamiseen on Strategy-suunnittelumalli, jossa järjestelmä sisältää erilaisia käyttäytymismalleja. Uusia malleja on tällöin myös yksinkertaista lisätä järjestelmään.

Itseohjautuvat järjestelmät voivat olla erittäin monimutkaisia ja ne on sen vuoksi suunniteltava huolellisesti. Jos järjestelmässä on paljon erilaista mukauttavaa toiminnallisuutta, on vaarana, että eri tekijät mukauttavat samoja järjestelmän osia. Tällöin järjestelmä ei välttämättä enää toimi niin

kuin sen pitäisi, mikäli se on suunniteltu huonosti. Eri mukauttamismekanismien tulee siis olla tietoisia siitä, miten muut mekanismit muuttavat järjestelmää, jotta toiminta pysyy johdonmukaisena.

Lisää itseohjautuvien järjestelmien suunnittelua ja toteutusta tukevia malleja ja menetelmiä kuitenkin tarvitaan. Koska itseohjautuvuus käsittää paljon erilaisia ominaisuuksia, tulisi selvittää, millaisilla malleilla ja mekanismeilla niitä voidaan toteuttaa. Tämä helpottaisi oleellisesti järjestelmäarkkitehtien työtä ja nopeuttaisi järjestelmien suunnittelua vähentäen näin järjestelmien kokonaiskustannuksia.

## Viiteluettelo

[Anderson et al., 2009] Jesper Anderson, Rogério de Lemos, Sam Malek and Danny Weyns, Modeling dimensions of self-adaptive software systems. In: B. H. C. Cheng et al. (eds.) *Software Engineering for Self-Adaptive Systems*. Dagstuhl Seminar, 2009, 27-47.

[Babaoglu et al., 2006] Ozalp Babaoglu, Geoffrey Canright, Andreas Deutsch, Gianni A. Di Caro, Frederick Ducatelle, Luca M. Gambardella, Niloy Ganguly, Márk Jelasity, Roberto Montemanni, Alberto Montresor and Tore Urnes, Design patterns from biology for distributed computing. *ACM Transactions on Autonomous and Adaptive Systems,* **1**, 1 (Sep 2006), 26-66.

[Brun and Medvidovic, 2007] Yuriy Brun and Nenad Medvidovic, An architectural style for solving computationally intensive problems on large networks. In: *Proc. of the International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2007, 2-9.

[Cheng et al., 2009] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi and Jeff Magee, Software engineering for self-adaptive systems: A research roadmap. In: B. H. C. Cheng et al. (eds.) *Software Engineering for Self-Adaptive Systems*. Dagstuhl Seminar, 2009, 1-26.

[Dean and Ghemawat, 2008] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, **51**, 1 (Jan 2008), 107-113.

[Erenkrantz et al., 2007] Justin R. Erenkrantz, Michael M. Grolick, Girish Suryanarayana and Richard N. Taylor, From representations to computations: The evolution of web architectures. In: *Proc. of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2007, 255-264.

[Eugster et al., 2003] Patrick T. Eugster, Pascal A Felber, Rachid Guerraoui and Anne-Marie Kermarrec, The many faces of publish/subscribe. *ACM Computing Surveys,* **35**, 2, (Jun 2003), 114-131.

[Fielding and Taylor, 2000] Roy T. Fielding and Richard N. Taylor, Principled design of the modern web architecture. In: *Proc. of the 22nd International Conference on Software Engineering*, 2000, 407-416.

[Gamma et al., 1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Garlan et al., 2004] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl and Peter Steenkiste, Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, **37**, 10 (Oct 2004), 46-54.

[Garlan and Schmerl, 2002] David Garlan and Bradley Schmerl, Model-based adaptation for self-healing systems. In: *Proc. of the First Workshop on Self-Healing Systems*, 27-32.

[Gomaa and Hussein, 2004] Hassan Gomaa and Mohamed Hussein, Software reconfiguration patterns for dynamic evolution of software architectures. In: *Proc. of the 4th Working IEEE/IFIP Conference on Software Architecture*, 2004, 79-88.

[Gorla, 2007] Alessandra Gorla, Towards design for self-healing. In: *Proc. of the Fourth International Workshop on Software Quality Assurance*, 2007, 86-89.

[Gorlick and Razouk, 1991] Michael M. Gorlick and Rami R. Razouk, Using weaves for software construction and analysis. In: *Proc. of the 13th International Conference on Software Engineering*, 1991, 23-34.

[Kramer and Magee, 2007] Jeff Kramer and Jeff Magee, Self-managed systems: An architectural challenge. In: *Future of Software Engineering 2007*, 259-268.

[Kuang and Ormandjieva, 2008] Heng Kuang and Olga Ormandjieva, Self-monitoring of non-functional requirements in reactive automatic systems framework: a multi-agent systems approach. In: *Proc. of the Third International Multi-Conference on Computing in the Global Information Technology*, 2008, 186-192.

[Medvidovic and Mikic-Rakic, 2001] Nenad Medvidovic and Marija Mikic-Rakic, Architectural support for programming-in-the-many. Technical report USC-CSE-2001-506. Available as *http://csse.usc.edu/csse/TECHRPTS/2001/usccse2001-506/usccse2001-506.pdf.*

[Menascé et al., 2010] Daniel A. Menascé, João P. Sousa, Sam Malek and Hassan Gomaa, QoS architectural patterns for self-architecting software systems. In: *Proc. of the 7th International Conference on Autonomic Computing*, 2010, 195-204.

[Menascé et al., 2011] Daniel A. Menascé, Hassan Gomaa, Sam Malek and João P. Sousa, SASSY: A framework for self-architecting service oriented systems. *IEEE Software,* **28**, 6 (Nov-Dec 2011), 78-85.

[Mühl et al., 2007] Gero Mühl, Matthias Werner, Michael A. Jaeger, Klaus Herrmann and Helge Parzyjegla, On the definitions of self-managing and self-organizing systems. In: *Proc. of the 2007 ITG-GI Conference of Communication in Distributed Systems (KiVS)*, 2007, 1-11.

[OMG, 2006] Object Management Group. http://www.omg.org. Checked 21.10.2012.

[OODesign, 2012] Object Oriented Design, Design patterns. http://www.oodesign.com/. Checked 8.12.2012.

[Oreizy et al., 1999] Peyman Oreizy, Michael M. Gorlick, Richard M. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum and Alexander L. Wolf, An architecture-based approach to self-adaptive software. *Intelligent Systems and Their Applications,* **14**, 3 (May/Jun 1999), 54-62.

[Oreizy et al., 2008] Peyman Oreizy, Nenad Medvidovic and Richard N. Taylor, Runtime software adaptation: Framework, approaches, and styles. In: *Companion of the 30th International Conference on Software Engineering*, 2008, 899-910.

[Ramirez and Cheng, 2010] Andres J. Ramirez and Betty H. C. Cheng, Design patterns for developing dynamically adaptive software. In: *Proc. of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2010, 49-58.

[Riva et al., 2006] Oriana Riva, Cristiano di Flor, Stefano Russo and Kimmo Raatikainen, Unearthing design patterns to support context-awareness. In: *Proc. of Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops*, 2006, 383-387.

[Salehie and Tahvildari, 2009] Mazeiar Salehie and Ladan Tahvildari, Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, **4**, 2 (May 2009), Article No. 14.

[Serugendo et al., 2004] Giovanna Di Marzo Serugendo, Noria Foukia, Salima Hassas, Anthony Karageorgos, Soraya Kouadri Mostéfaoui, Omer F. Rana, Mihaela Ulieru, Paul Valckenaers and Chris Van Aart, Self-organization: paradigms and applications. In: G. Di Marzo Serugendo, et al. (eds.) *Engineering Self-Organizing Systems, LNCS* **2977**, Springer 2004, 1-19.

[Souza, 2012] Vítor E. Silva Souza, A requirements-based approach for the design of adaptive systems. In: *Proc. of 34th International Conference on Software Engineering*, 2012, 1635-1637.

[Taylor et al., 1995] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr. and Jason E. Robbins, A component- and message-based architectural style for GUI software. In: *Proc. of 17th International Conference on Software Engineering*, 1995, 295-304.

[Vromant et al., 2011] Pieter Vromant, Danny Weyns, Sam Malek and Jesper Anderson, On interacting control loops in self-adaptive systems. In: *Proc. of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2011, 202-207.

[Welsh et al., 2011] Kristopher Welsh, Peter Sawyer and Nelly Bencomo, Towards requirements aware systems: run-time resolution of design-time assumptions. In: *Proc. of the 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, 560-563.

[Weyns and Holvoet, 2007] Danny Weyns and Tom Holvoet, An architectural strategy for self-adaptive systems. In: *Proc. of the 2007 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2007, 3-12.

[Weyns et al., 2012] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Anderson, Holger Giese and Karl Göschka, On patterns for decentralized control in self-adaptive systems. In: R. de Lemos, et al. (eds.) *Self-Adaptive Systems, LNCS* **7475**, Springer 2012, 76-107.